Analiza zmogljivosti oblačnih in strežniških storitev

Uredil prof. dr. Miha Mraz

Maj 2017

Kazalo

Predgovor

Pričujoče delo je razdeljeno v deset poglavij, ki predstavljajo analize zmogljivosti nekaterih tipičnih strežniških in oblačnih izvedenk računalniških sistemov in njihovih storitev. Avtorji posameznih poglavij so slušatelji predmeta *Zanesljivost in zmogljivost računalniških sistemov*, ki se je v štud.letu 2016/2017 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 2017

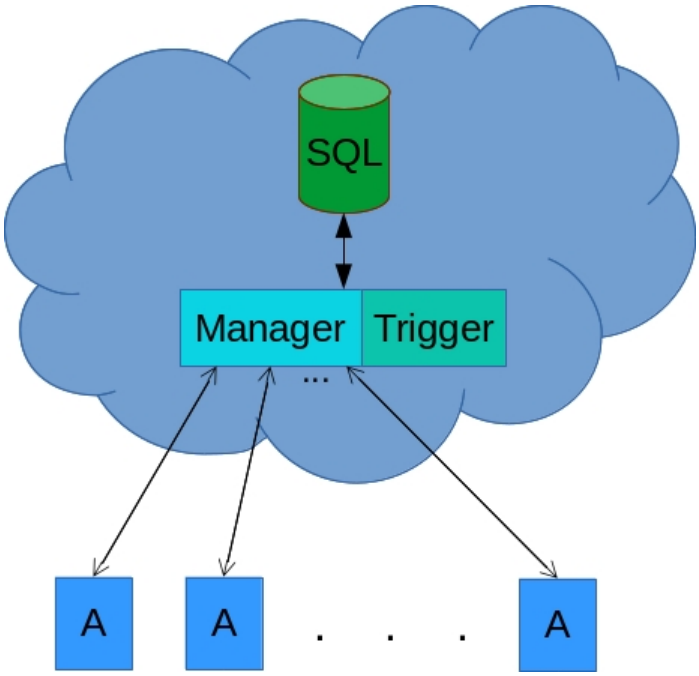
Poglavje 1

Testiranje SQL baze z Digital Ocean spletnim servisom

Adrian Jarc, Marko Kočevar,
Veronika Omahen, Miha Mencin

1.1 Opis problema

V pričujočem poglavju opišemo delovanje obsežne SQL baze postavljene z Digital Ocean spletnim servisom [1]. Zanima nas predvsem kako se bo baza odzivala na mnogo sočasnih zahtev za dostop do podatkov po vzoru električnih števcev [2] , [3].
Za namen testiranja smo postavili bazo v oblaku in spisali vmesnik za delo z bazo in agenti, ki komunicirajo z vmesnikom. Prožilec, ki je del vmesnika, bo pognal proces polnjenja baze, nato pa bo vmesnik poslal zahtevo agentom, da mu ti generirajo podatke za vstavljanje v bazo po različnih scenarijih. Osnovno shemo delovanja sistema vidimo na sliki 1.1.



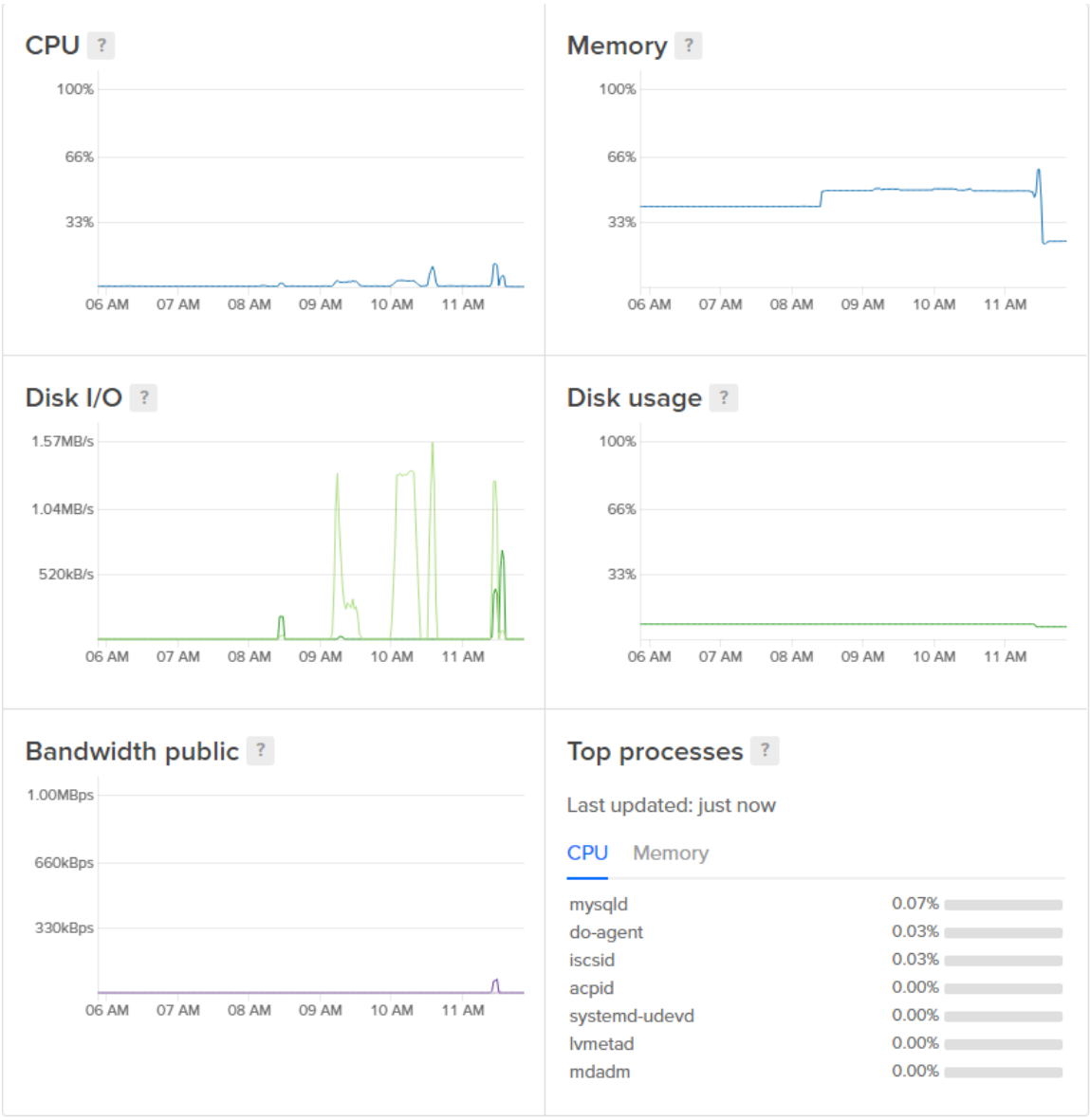
Slika 1.1: Osnovna shema delovanja. V oblaku imamo postavljeno bazo (MySQL) in vmesnik (Manager), preko katerega komuniciramo z bazo. Zraven je tudi sprožilec (Trigger), ki sproži pisanje v bazo oz. natančneje iz RAM-a na disk.

1.2 Izbira ponudnikov in tehnologij

Za ponudnika oblačnih storitev smo si izbrali Digital Ocean [1], predvsem zaradi enostavnosti uporabe in postavitve strežnika. Za bazo smo izbrali MySQL podatkovno bazo [4], saj je odprtokodna in jo je lahko postaviti na Linux strežniku. Sam vmesnik za komuniciranje z bazo in klient, ki simulira tisoče agentov oziroma števcev je spisan v Python programskem jeziku, za komunikacijo z bazo pa smo uporabili MySQLdb knjižnico [5]. Za testiranje nam Digital Ocean ponuja monitoring in diagnostiko, ki ju bomo uporabili za namene testiranja. Primer kako ta monitoring izgleda, lahko vidimo na sliki 1.2.

1.2.1 Tehnologije - Python

Vmesnik in klient smo implementirali v visoko-nivojskem jeziku Python. Za Python smo se odločli zato, ker ima velik nabor knjižnic in omogoča lažjo implementacijo naloge. Omogoča tudi lahko namestitev paketov s programom za upravljanje paketov PIP-om. Ker se Python tolmači, nam tudi omogoča lahko prenosljivost kode na druge platforme.



Slika 1.2: Izgled monitoringa, ki ga ponuja Digital Ocean.

1.2.2 Tehnologije - MySQL

Za podatkovno bazo smo izbrali MySQL, ki se nam je zdela najlažja za uporabo in je zaradi svoje odprtokodnosti najprimernejša za implementacijo na Linux strežniku.

1.2.3 Tehnologije - Ponudnik obačne storitve

Za najem strežnika v oblaku smo izbrali Digital Ocean, ker se nam je zdela najcenješa opcija in najbolj primerna za našo nalogo. Na njihovem sistemu teče operacijski sistem Linux. Specifikacije so navedene v naslednjem razdelku 1.2.4. Zaradi jasnosti bomo za virtualni strežnik uporabljali izraz virtualka. Digital Ocean tudi ne zaračunava dodatnih stroškov za lokacijo virtualke, zato smo lahko brez doplačila izbrali, da je naša virtualka locirana v Frankfurtu.

1.2.4 Specifikacije strežnika

Pri ponudniku smo se odločili za najcenejšo rešitev, saj je dovolj močna za testiranje delovanja. Specifikacije so sledeče:

- procesor: 1 jedro, Intel Xeon CPU E5-2650L v3 @ 1.8GHz,
- ram: 1GB,
- disk: 30GB.

1.3 Baza in breme

Sama baza ima eno tabelo z atributom *ID*, ki enolično določa posameznega agenta in množico atributov, ki predstavljajo minutne intervale meritev, tipa int, saj za testiranje ne potrebujemo večje natačnosti. Ker testiramo, koliko podatkov lahko zapišemo v bazo v najkrašem času, je naše breme število podatkov, ki jih pošiljajo klienti v oblak.

Breme bi lahko obravnavali tudi kot število zahtevkov, ki pridejo na časovno enoto, katere lahko strežnik obdela. Vendar, ker je težko zagnati zadostno število klientov, smo se raje odločili za pristop, kjer povečujemo velikost poslanih podatkov. Dodatne metrike zmogljivosti bomo lahko pridobili z meritvami.

1.4 Metrike

Za metriko bomo vzeli hitrost zapisovanja v bazo.

1.5 Prepustnost

Tukaj bomo merili zmožnost strežnika, kako hitro lahko zapisuje podatke v bazo. To bomo naredili tako, da bomo pošiljali veliko število podatkov na strežnik.

Možno bi tudi bilo, da ne bi zaradi konfiguracije strežnika bili zmožni zapisati vseh podatkov v bazo.

1.6 Meritve - časi

Podatki: podatki, ki smo jih pošiljali za prve 3 meritve so istega velikostnega razreda, tako da lahko primerjamo rezultate meritev. Pošiljali smo samo eno celoštevilsko vrednost, ki je bila zapisana kot niz.

1.6.1 Meritev 1: Osnovni model

- **Opis poskusa:** Osnovni model komuniciranja med vmesnikom in agenti je zaporedni način. Vmesnik pošlje aplikaciji, katera simulira agente, zahtevo za vrednost meritve, ter nato počaka na odgovor, katerega takoj shrani v bazo. Poslali smo le en podatek (eno celo število zapisano kot niz). Imeli smo 1000 agentov. Komunikacija med vmesnikom v oblaku in lokalno aplikacijo poteka preko vtičnice.
- **Hipoteza:** Pričakujemo, da bo aplikacija delovala brez večjih problemov.
- **Robni pogoji:** Meritve so bile izvedene 29.3.2017 ob 18.00 v Ljubljani.
- **Rezultati meritev:** Iz rezultatov v tabeli 1.1 vidimo, da je naša aplikacija povprečno potrebovala 47,55s, da je zapolnila en interval meritev tisočih agentov v bazi. To je povprečni čas 50 meritev. Standardni odklon je bil 2s.
- **Komentar:** Ker so to bile prve meritve nimamo kaj veliko za dodati, saj še nimamo podatkov za primerjanje.

1.6.2 Meritev 2: Osnovni model - posodobljen

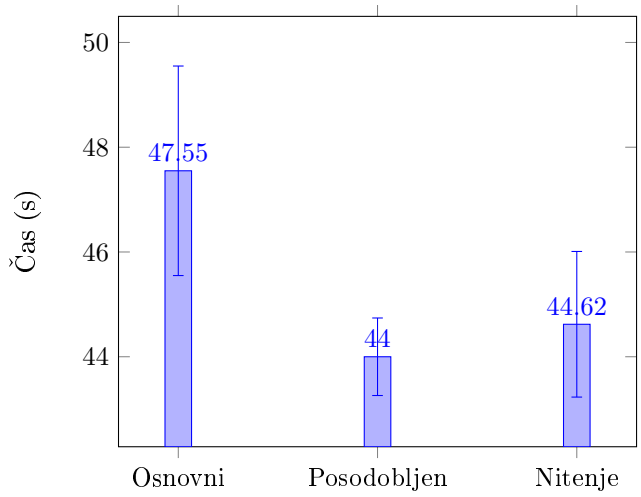
- **Opis poskusa:** Osnovni model smo za malenkost izboljšali tako, da si vmesnik naprej shrani rezultate stotih meritev in jih nato zapiše vse v bazo.
- **Hipoteza:** Pričakujemo, da bomo imeli malo boljši čas, saj bomo zapisali več podatkov v bazo naenkrat in tem tudi bolje izkoristili predpomnilnik na SSD disku.
- **Robni pogoji:** Meritve so bile izvedene 29.3.2017 ob 18.43 v Ljubljani.
- **Rezultati meritev:** S to posodobitvijo smo pridobili povprečno 3,55s, standardni odklon pa se je zmanjšal na 0,74s. Tukaj smo znova imeli 50 meritev.
- **Komentar:** Čas smo za malo izboljšali, kar je bilo za pričakovati. Sprememba ni velika, saj naš sistem teče na SSD disku.

1.6.3 Meritev 3: Osnovni model - niti

- **Opis poskusa:** Ker so bili rezultati še vedno slabi, smo poskusili vmesnik implementirati s pomočjo niti. Ideja je bila, da bi vmesnik deloval po načelu proizvajalec-porabnik. Ena nit izključno skrbi za komunikacijo z agenti in dobljene meritve shranjuje v globalno listo meritev. Druga nit je zadolžena za branje shranjenih vrednosti in vpis le-teh v bazo.
- **Hipoteza:** Pričakujemo, boljši čas kot pri 1.6.1 in 1.6.2.
- **Robni pogoji:** Meritve so bile izvedene 30.3.2017 ob 12.13 v Ljubljani.
- **Rezultati meritev:** S posodobitvijo smo dosegli povprečni čas 44,62s in standardnim odklonom 1,39. Rezultati so prikazani v tabeli 1.1 in sliki 1.6.3.
- **Komentar:** Čas smo nekoliko zmanjšali glede na 1.6.1 (Osnovni model), vendar se je prejšnja meritev 1.6.2 (Osnovni model - posodobljen) izkazala za hitrejšo.

Model	Povprečničas (s)	Standardni odklon (s)
Osnovni (1.6.1)	47,55	2,00
Posodobljen (1.6.2)	44,00	0,74
Nitenje (1.6.3)	44,62	1,39

Tabela 1.1: Rezultati meritev osnovnega modela in njegovih izboljšav.



Slika 1.3: Grafični prikaz vrednosti tabele 1.1.

1.6.4 Meritev 4: Pomično okno

- **Opis poskusa:** Osnovni model zaporedne komunikacije smo zamenjali s simulacijo pomičnega okna. Pomično okno je pri naši implementaciji v bistvu medpomnilnik, ki lahko naenkrat hrani določeno število zahtev/odgovorov. Vmesnik pošlje zahtevo za meritve 2, 4, 10, 20 števcev hkrati, dobi odgovor in nato rezultate vpiše v bazo. Za ta pristop smo se odločili, ker nam z našimi sredstvi ni bilo mogoče simulirati na tisoče agentov, ki bi sočasno dostopali do baze.

- **Hipoteza:** Predvidevamo, da bomo močno izboljšali pretočnost podatkov, ker ne bomo zaporedno pošiljali podatkov in čakali na vsakega agenta posebej.

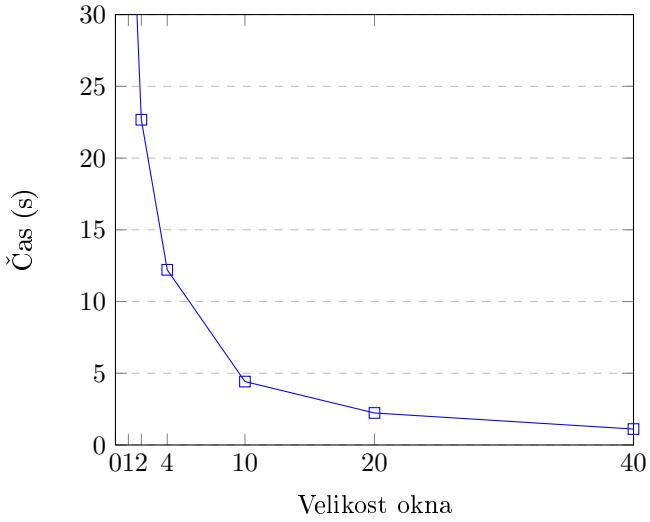
- **Robni pogoji:** Meritve so bile izvedene 12.4.2017 ob 12.13 v Ljubljani.

- **Rezultati meritev:** Rezultate meritev vidimo v tabeli 1.2 in na sliki 1.4. Iz njih lahko vidimo, da se je čas vpisa enega intervala meritev močno zmanjšal.

- **Komentar:** Iz rezultatov lahko sklepamo, da je problem predvsem v času potrebnem za komunikacijo med vmesnikom in agenti, medtem ko je sam vpis v bazo oziroma posodabljanje vrednosti izjemno hitra operacija. Pri nadaljnih testih se je pokazalo, da tudi pri velikosti okna 250, aplikacija še vedno brez težav zapiše vrednosti pravočasno v bazo. Ideja za naprej je stestirati druge načine komunikacije med vmesnikom in aplikacijo, ki simulira agente, saj se tukaj izgubi največ časa. Tudi naša implementacija z vtičnico lahko da ni najbolj primerna za ta problem.

Velikost okna	Povprečničas (s)	Standardni odklon (s)
2	22,67	1,42
4	12,21	1,03
10	4,42	0,47
20	2,23	0,26
40	1,11	0,19

Tabela 1.2: Povprečni časi 50 meritev pomičnega okna.



Slika 1.4: Povprečeni rezultati 50 meritev glede na velikost pomičnega okna.

Kot vidimo, so meritve implementirane s premičnim oknom veliko hitrejše, od 2-krat do 20-krat, kot z drugimi tehnikami, ki smo jih uporabili.

1.7 Meritve - hitrost pisanja v bazo

1.7.1 Meritev 5: pomično okno - hitrost pisanja v bazo

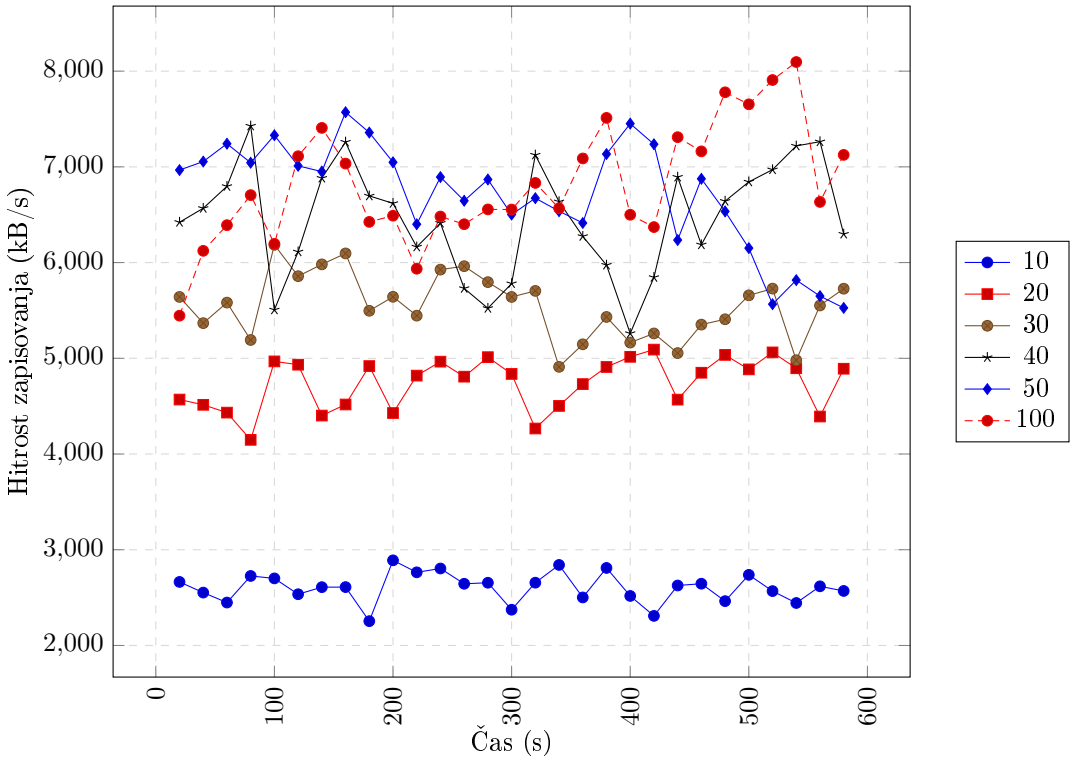
- **Opis poskusa:** Z verzijo vmesnika, ki uporablja pomično okno, smo dodatno testirali zmogljivost zapisovanja podatkov na disk glede na velikost pomičnega okna. Za to verzijo programa smo se odločili, ker je bil prenos podatkov od agentov do vmesnika najhitrejši in je s tem tudi najbolj primeren za naš test. Obremenjenost diska smo merili s programom za monitoring sistema *nmon* [6].

- **Hipoteza:** Če bomo poslali zadostno število podatkov, bi moralo sistemu zmanjkati RAM-a in bi se tako mogel odpovedati oziroma aplikacija bi se morala zapreti. Druga možnost je, da nimamo dovolj močne linije od agentov do vmesnika in do tega nikoli ne bo prišlo.

- **Robni pogoji:** Meritve so bile izvedene 21.5.2017 ob 15.33 v Ljubljani.

- **Rezultati meritev:** Na sliki 1.5 vidimo s kakšno hitrostjo je vmesnik zapisoval dobljene podatke. Za posamezno velikost okna smo deset minut beležili hitrost zapisovanja v sekundnih intervalih. Meritve smo nato povprečili v dvajset sekundnih intervalih, saj je hitrost pri manjši velikosti okna precej varirala.

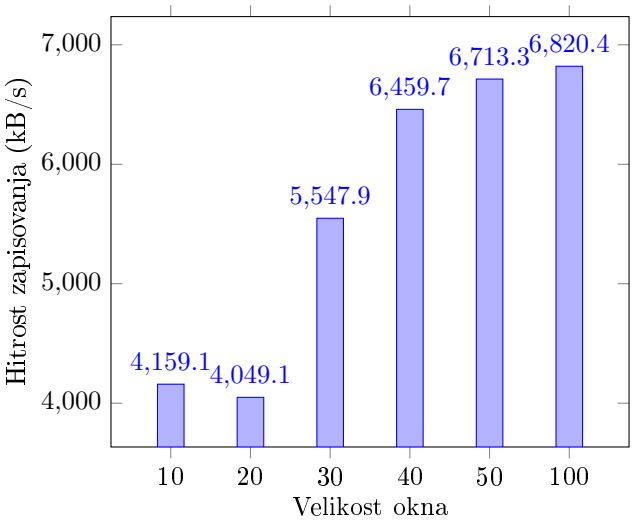
- **Komentar:** Vidimo, da pri velikosti okna 10 in 20, zmankuje podatkov, zato disk ni bil popolnoma izkoriščen. To se vidi iz slike 1.5, kot tudi 1.6. S slike 1.6, kjer smo povprečili vse meritve glede na velikost okna. Merili smo 10 minut, kar je znašalo 577 meritev. Z večanjem okna se nato tudi povečuje povprečna hitrost zapisovanja na disk, kar se tudi vidi iz prej navedenega grafa.



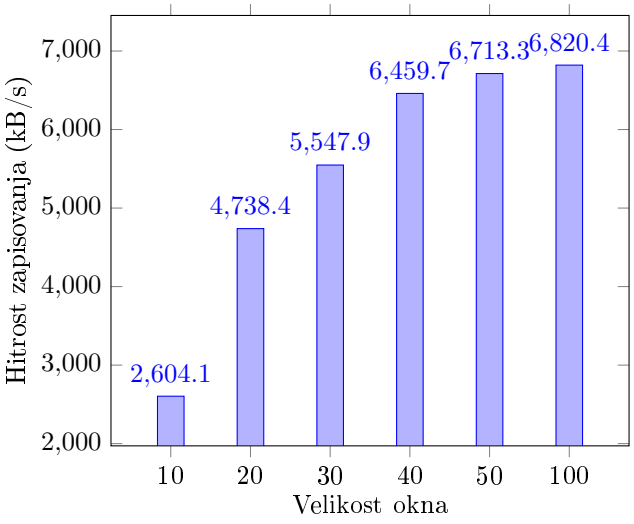
Slika 1.5: Graf hitrosti zapisovanja na disk glede na velikost okna.

Za boljšo predstavo, kako se je hitrost pisanja v bazo spremijala smo povprečili podatke z zgornjega slike 1.5 glede na velikost.

Tu se lepo vidi, da pri dovolj velikem oknu, hitrost zapisovanja v bazo narašča. Pri teh testih smo prišli do maksimalne povprečne hitrosti okoli 7MB/s. K tej meji se približamo pri velikosti okna 40, pri velikosti 50 in 100 pa je že prihajalo do večje porabe pomnilnika, saj strežnik ni bil dovolj hiter, da bi pravočasno zapisoval vse vrednosti iz pomnilnika na disk. Porabo pomnilnika vidimo na grafu 1.10, kjer sta vidna dva manjša vrha malo pred 16.00. Opazimo pa nekaj čudnega na grafu. Vidimo, da je prepustnost diska pri oknu velikosti 20 manjša kot pri oknu velikosti 10. Ker so rezultati čudni, jih bomo ponovili



Slika 1.6: Povprečene hitrosti zapisovanja na disk.



Slika 1.7: Povprečene hitrosti zapisovanja na disk, ponovili meritve za velikost okna 10 in 20.

za ti dve velikosti.

Po ponovnih meritvah vidimo, da se zdaj rezultati ujemajo z našimi predpostavkami. Zakaj je prislo do take anomalije se sprašujemo? Problem bi lahko bil v našem načinu merjenja hitrosti zapisovanja na disk. Program *nmon* [6] smo zagnali tako, da nam je na vsake pol sekunde izpisal hitrost zapisovanja na disk. Imeli smo nesrečo in je pri merjenju hitrosti zapisovanja na disk pri velikosti okna 20 velikokrat zajel ob času, ko je bilo zapisovanje majhno.

1. Hipoteza: Mislimo, da vmesnik ni sposoben hitreje zapisovati podatkov v bazo. To je lahko pogojeno z Python-nom,

2. Hipoteza: Knjižnica, ki jo uporabljamo za komunikacijo, med bazo in vmesnikom, po vsej verjetnosti ni dovolj hitra.

1.7.2 Meritev 6: Točka odpovedi

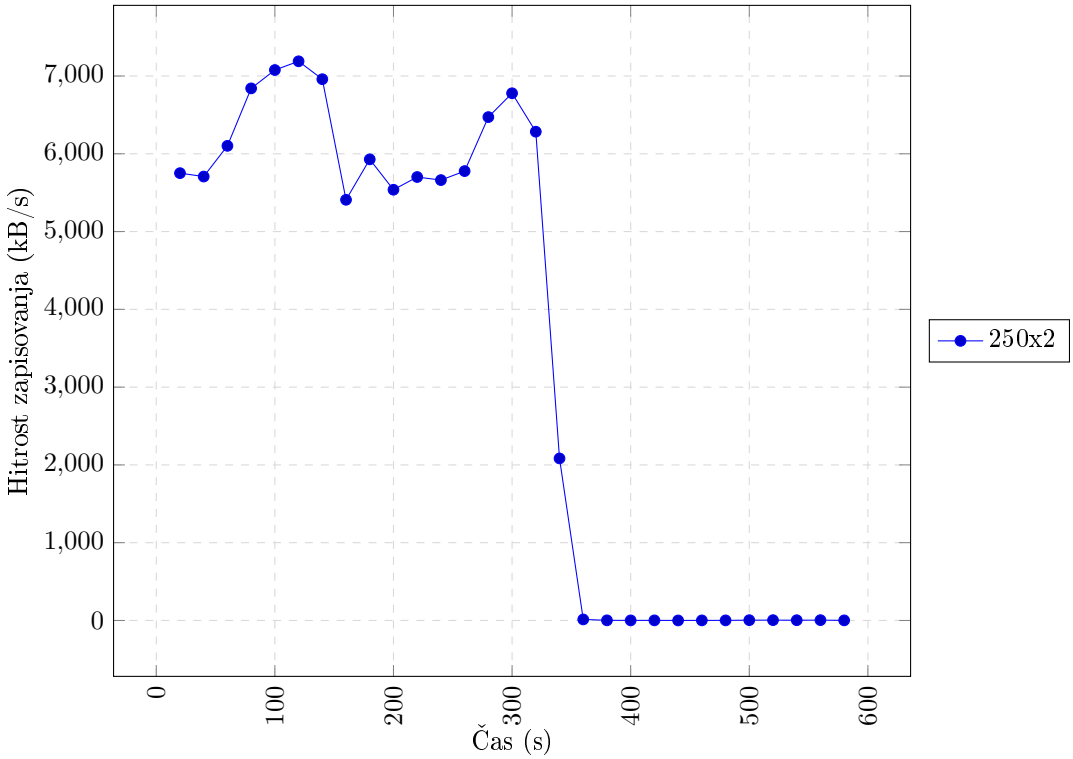
- **Opis poskusa:** Tukaj smo poskusili najti točko odpovedi sistema. Do odpovedi bi lahko prišlo že pri velikostih okna nad 50, saj se že tam pojavi pomanjkanje pomnilnika. Da bi postopek pohitrili, smo zagnali dve niti za komunikacijo z velikostjo okna 250 in le eno za zapis v bazo.

- **Hipoteza:** Sistem bo odpovedal oziroma aplikacija se bo zaprla, ko bo zmanjkalo pomnilnika.

- **Robni pogoji:** Meritve so bile izvedene 21.5.2017 ob 15.33 v Ljubljani.

- **Rezultati meritev:** Na sliki 1.8 vidimo, da aplikacija normalno deluje približno 300 sekund, nato hitrost pisanja naglo upade. Uspelo nam je porušiti sistem.

- **Komentar:** Po pričakovanjih, nit za zapis ni zmogla vpisati vrednosti v bazo, katere so se nato nabirale v pomnilniku. Ker smo zagnali dve niti za komunikacijo je poraba pomnilnika strmo naraščala, kot vidimo na sliki 1.10, na kateri je velik vrh kmalu po 16.00. V trenutku ko je bil zaseden cel pomnilnik, je aplikacija postala neodzivna in nehala delovati, kot vidimo na sliki 1.8. Po 300ih sekundah pade hitrost pisanja na nič. V tem trenutku je sistem začel z maksimalno hitrostjo brati po disku, vidno na sliki 1.11, kjer je velik vrh kmalu po 16.00. Po krajšem času neodzivnosti, se je vmesnik terminiral in sprostil podatke v pomnilniku. Predvidevamo, da je sistem sam zaprl aplikacijo, da je zaščitil sebe. To sklepamo po tem, ker nam ni bilo treba na novo zaganjati sistema.



Slika 1.8: Graf na keterem vidimo točko odpovedi sistema.

Slika 1.9: Graf porabe procesorja v času poganjanja testov za meritve 1.7.1 in 1.7.2.

Slika 1.10: Graf porabe pomnilnika v času poganjanja testov za meritve 1.7.1 in 1.7.2.

Slika 1.11: Graf delovanja diska v času poganjanja testov za meritve 1.7.1 in 1.7.2.

1.8 Zaključek

Tekom seminarske naloge smo testirali, kako se bo baza odzivala na mnogo sočasnih zahtev za dostop do podatkov po vzoru električnih števcev [2], [3].
Ugotovili smo, da bi za potrebe posameznika ali manjših poslovnih subjektov naša rešitev prišla prav ([2], [3]). Lahko bi zajemali tudi meritve drugih senzorskih naprav, ki imajo dostop do medmrežja.
Naša rešitev bi bila lahko primerna tudi za večje korporacije, vendar bi potrebovali bolj zmogljive strežnike, za katere pa sredstev žal nismo imeli. Mislimo da je problem skalabilen in da bi lahko z večanjem kapacitet strojne opreme povečali tudi število odjemalcev. Pri nas se je za ozko grlo sistema izkazal disk. Pri večjem stevilu odjemalcev bi se pa lahko pojavili dodatni problemi, na katere nismo naleteli med našim testiranjem. Če bi imeli dovolj sredstev bi lahko postavili strežnik tudi lokalno. Uporabljali bi zmogljivejše SSD diske z RAID0/RAID10 [7] konfiguracijo.

Poglavje 2

Analiza zmogljivosti sporočilnega sistema v oblaku

Greta Gašparac, Jan Hartman, Matija Kljun, Rok Pajnič

2.1 Opis oblačne storitve

V pričujočem poglavju zmogljivostno analiziramo sporočilni sistem, ki deluje na osnovi protokola MQTT. Podrobneje - v oblak smo namestili posrednika (angl. *broker*) sporočil MQTT in z različnimi testi metrik opazovali njegovo zmogljivost. Našo storitev smo namestili na različne ponudnike oblačnih storitev.

2.2 Izbira tehnologij

Pri izbiri tehnologij smo hoteli izbrati takšne, ki niso prestare, so dandanes v uporabi in niso prezahtevne za uporabo. Namestitev v oblak in povezovanje s storitvijo naj bi zahtevala minimalno količino dela, da bi se lažje posvetili nalogam v sklopu tega predmeta. Protokol MQTT je po teh kriterijih še posebej izstopal, saj je z naraščanjem števila IoT naprav čedalje bolj vseprisoten.

2.2.1 MQTT

MQTT je ISO standard, ki je zasnovan na aplikacijski plasti. Za uporabo potrebujemo posrednika, ki distribuira sporočila naročnikom glede na kanal, na katerega so naročeni. Prednost tega protokola je njegova kompaktnost in malo režijskih stroškov (angl. *overhead*), zato je primeren tudi za večje sisteme. Pogosto je uporabljen na področju IoT, uporablja pa ga celo Facebook v svoji

aplikaciji Messenger.

MQTT deluje po vzorcu *publish/subscribe*, ki je alternativa tradicionalnemu modelu strežnik/odjemalec. Naročnik in pošiljatelj ne vesta eden za drugega, za vse skrbi tretja komponenta - *broker* oz. posrednik.

Glavne značilnosti tega načina:

- ni nujno, da se naročnik in pošiljatelj poznata,
- ni nujno, da naročnik in pošiljatelj tečeta istočasno,
- operacije na obeh niso ovirane zaradi prejemanja ali pošiljanja (asinhronost delovanja).

Klient

MQTT klient je lahko vsaka naprava, od mikrokrmilnika pa do strežnika, ki ima nameščeno MQTT knjižnico in je preko nekega omrežja povezana na MQTT posrednika. Knjižnice so na voljo za veliko programskih jezikov, kot so Java, JavaScript, Python, Go, .NET itd.

Posrednik

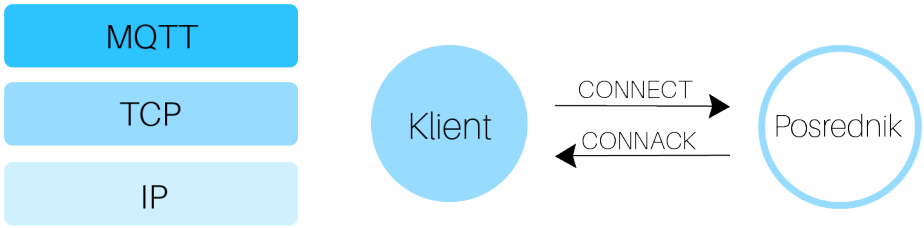
Posrednik predstavlja srce vsakega *publish/subscribe* protokola. Odgovoren je za prejemanje, filtriranje in pošiljanje sporočil ustreznim naročnikom. MQTT posrednik ima t.i. *subject-based filtering*, kar pomeni, da filtrira glede na *topic* oz. kanal, ki je del vsakega sporočila. Klient se naroči na kanal in od posrednika dobi vsa sporočila, ki so poslana nanj.
Posrednik prav tako skrbi za avtentikacijo in avtorizacijo klientov. Pomembno je, da je skalabilen, zlahka integriran v zaledne sisteme, enostaven za upravljanje in zanesljiv.

MQTT povezava

MQTT je baziran nad TCP/IP plastjo (slika 2.1). Povezava se inicializira, ko klient pošlje CONNECT zahtevek posredniku. Ta se odzove z odgovorom CONNACK in statusno kodo. Ko je povezava vzpostavljena, jo bo posrednik obdržal vse dokler klient ne pošlje ukaza za prekinitev ali dokler na kak drug način ne izgubi povezave.

Quality of Service

Quality of Service je nastavljiv parameter naše komunikacije, ko govorimo o garanciji prejetja sporočila. Poudariti je treba, da ločimo kvaliteti pošiljanja pošiljatelj-posrednik in posrednik-prejemnik (parameter QoS torej ni nujno enak). MQTT ima 3 različne stopnje:



Slika 2.1: Prikaz protokola v modelu OSI in inicializacije povezave klienta s posrednikom.

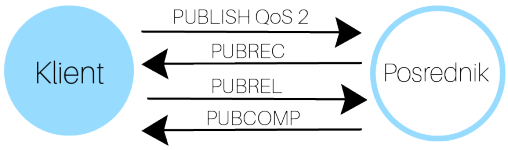
- QoS = 0: Princip *fire and forget*, kjer pošiljatelj pošlje sporočilo in se ne zanima za to, če so ga naslovniki prejeli. Shemo komunikacije lahko vidimo na sliki 2.2.

Slika 2.2: Prikaz komunikacije z nastavitvijo $qos = 0$.

- QoS = 1: Naročniki sporočilo prejmejo vsaj enkrat, lahko pa tudi večkrat. Pošiljatelj hrani sporočilo, dokler od prejemnika ne dobi odziva PUBACK. Če PUBACK ne prispe v nekem razumnem času, pošiljatelj ponovno pošlje sporočilo. Shemo komunikacije lahko vidimo na sliki 2.3.

Slika 2.3: Prikaz komunikacije z nastavitvijo $qos = 1$.

- QoS = 2: Vsako sporočilo je prejeto natanko enkrat. Ta način je najbolj varen in posledično najpočasnejši. Prejemnik prejme sporočilo in se odzove s PUBREC, še vedno pa hrani referenco na sporočilo. Ko pošiljatelj prejme PUBREC, lahko pozabi na začetno sporočilo, ker ve, da ga je naslovnik uspešno prejel. Shrani si PUBREC in odgovori s PUBREL. Ko prejemnik to dobi, lahko pozabi na vsa shranjena stanja, odgovori pa s PUBCOMP. Ob prejemu tega lahko na vsa stanja pozabi tudi pošiljatelj. Shemo komunikacije lahko vidimo na sliki 2.4.



Slika 2.4: Prikaz komunikacije z nastavitvijo $qos = 2$.

Shranjevanje sporočil

Da posrednik shrani sporočila morata biti izpolnjena 2 pogoja: klient mora imeti vklopljeno nenehno sejo (angl. *persistent session*) in parameter QoS mora biti večji od 0.
Neskončna seja: ko se klient poveže z MQTT posrednikom, se naroči na vse želene kanale. Ko se naslednjič poveže, mora to storiti ponovno. To je v nekaterih primerih zoprno, zato lahko klient vklopi nenehno sejo in posrednik shrani vse relevantne podatke o klientu (sejo, ID, naročnine, sporočila, ki jih še ni potrdil). Tako lahko klienti prejmejo sporočila, tudi če v času pošiljanja niso bili prijavljeni [8].

2.2.2 Oblačne storitve

Pri izbiri oblačnih storitev je dandanes možnosti ogromno. Sami smo se odločili za DigitalOcean predvsem zaradi enostavne uporabe ter možnosti uporabe brez vlaganja lastnega denarja, z brezplačnimi krediti pridobljenimi v GitHub Education Pack [9]. Poleg DigitalOcean smo pri nekaterih testiranjih preiskusili tudi oblačne storitve Amazon Web Services (AWS).

DigitalOcean

DigitalOcean je ponudnik oblačne infrastrukture. Podatkovne centre ima po vsem svetu in je od vseh izbranih ponudnikov najpreprostejši za hitro postavitev virtualke (dropleta) z izbrano distribucijo Linuxa ali FreeBSD. Na žalost ne nudi zastonj poizkusne dobe, na našo srečo pa smo v GitHub Education Packu dobili kodo za 50$ kredita. Glede strojne moči smo bolj omejeni kot drugje - tu ni možno prilagoditi vsake komponente posebej, temveč le celoto (več procesorskih jeder - več pomnilnika - višja cena) [10].

Amazon Web Services

Amazon Web Services (AWS) je oblačna storitev podjetja Amazon. Brezplačno omogoča 12 mesecev omejene uporabe s programom AWS Free Tier, s katerim imamo na voljo 750 ur brezplačne uporabe Linux ali Windows virtualke na instanci t2.micro z 1GiB spomina in 1 vCPU. Študentom poleg tega omogoča brezplačne kredite ($75 - $100), ki omogočajo neomejeno uporabo. S pomočjo promocijske kode Github edukacijskega paketa smo pridobili še dodatnih $50

kreditov. S temi krediti lahko prosto izbiramo med danimi instancami in si strojno moč prilagodimo glede na naše potrebe. Instance t2.micro poganja procesor družine Intel Xeon, ki omogoča tako imenovana *burst* obdobja, v katerih dosežejo frekvenco delovanja do 3.3 GHz [11].

Izbrane zmogljivosti oblačnih sistemov

V želji po točnih in primerljivih rezultatih smo poskušali izbrati čim bolj performančno enakovredne oblačne virtualne sisteme. Vsi izbrani sistemi nam dajejo na razpolago **1 virtualno procesno enoto** in **1GiB delovnega pomnilnika**. Kot lokacijo strežnikov smo v obeh primerih izbrali Evropo. Te storitve so nam na voljo po naslednjih cenah:

- Amazon Web Services: 0.014$ / h,
- Digital Ocean: 10$ / mesec.

2.3 Breme

Ker smo se ukvarjali s sporočilnim protokolom, je tudi naše breme predstavljalo besedilno sporočilo oziroma množica sporočil, ki so bila posredovana od sporočevalca do MQTT posrednika na oblaku in od tam naprej do naročnikov. Lahko bi rekli da so breme našega sporočilnega sistema pravzaprav vsi aktivni uporabniki. To smo tudi testirali.

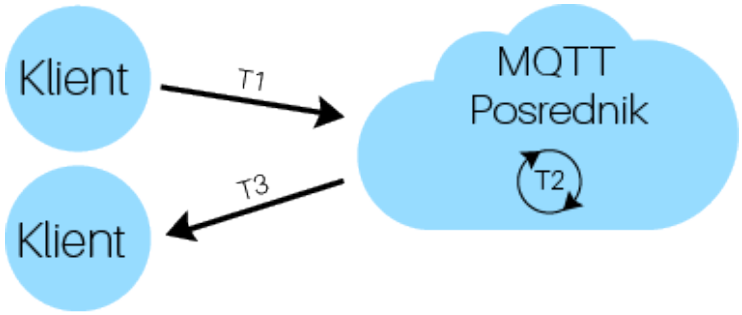
V začetni fazi smo testirali najosnovnejšo, minimalno obliko sporočanja - en sporočevalec in en naročnik na istem kanalu. Temu so sledile meritve na različnih lokacijah, kjer smo opazovali razliko v omrežjih. Nadaljevali smo z razširjanjem različnih parametrov, kot so število klientov in kanalov, frekvenca pošiljanja sporočil, dolžine sporočil. S temi različnimi scenariji smo se skušali približati realnejšemu bremenu in opazovali, kako se oblak obnaša in kako te spremembe zaznavajo klienti, ki uprabljajo oblačno storitev. Breme smo testirali tudi z različnimi konfiguracijami MQTT posrednika.

2.4 Metrike

Kot metrike zmogljivosti sistema bomo uporabili čas (merjenje času, potrebnega za proces komunikacije), prepustnost (merjenje števila sporočil, ki jih posrednik prepošlje v neki časovni enoti) in uporabniško izkušnjo (kako počasi še lahko dela sistem, preden se le-ta poslabša).

2.4.1 Čas

Osredotočili se bomo na merjenje časa T. To je čas, ki je potreben za celoten proces komunikacije. Definiran je kot vsota časov T_1 (čas potovanja sporočila do oblaka), T_2 (čas obdelave sporočila na MQTT posredniku) in T_3 (čas potovanja sporočila do naročnika). Shema je na sliki 2.5.



Slika 2.5: Shema enostavnega komuniciranja med dvema klientoma.

Lokalnost

Za prvo poskusno meritev smo nastavili posrednika na oblaku *DigitalOcean*. Klient za pošiljanje (angl. *publisher*) je tekel v Kočevju in je z intervalom ene sekunde poslal sto sporočil, ki so vsebovale čas pošiljanja. Sporočila so prejemali trije klienti za prejemanje (angl. *subscribers*), in sicer v Ljubljani, Ribnici in Novi Gorici. Ob prejetem sporočilu so na podlagi trenutnega časa izračunali čas pošiljanja. Za sinhronizacijo časov na vseh klientih smo uporabili knjižnico *ntplib*, s katero smo izvajali poizvedbe za čas na Arnesov strežnik (ntp1.arnes.si). Poskus smo izvedli z različnimi nastavitvami *quality of service (qos)*. Merili smo celoten čas T.

Časi T_i

Ker nimamo dostopa do kode posrednika, žal ne moremo opazovati časov v obliki nekega *logginga*, zato smo se morali znajti. Poleg posrednika smo na oblak naložili še skripto pošiljatelja in prejemnika in merili T_2 na ta način. Ker se komunikacija tako izvaja precej lokalno, lahko rečemo, da sta časa T_1 in T_3 praktično 0 in ju lahko zanemarimo. Izračunali smo ju šele kasneje, ko smo enak test pognali tako, da je bil posrednik na oblaku, klienta pa na naših računalnikih. Zaradi majhnega bremena merimo čas 10 poslanih sporočil.

2.4.2 Prepustnost

V tej fazi smo testirali, kako se MQTT posrednik v oblaku obnaša ob različnih obremenitvah klientov in kanalov. Testni scenarij smo razdelili na dva ločena dela. V prvemu smo imeli različno število klientov, ki so bili vsi na enemu kanalu. Merili smo čas dostave sporočila do klientov in sicer na vseh treh nivojih kvalitete pošiljanja(QoS). V drugem scenariju smo pošiljali sporočila na več kanalov in pri vsakem poskusu povečali število kanalov. Tudi ta poskus smo opravili na vseh treh nivojih kvalitete pošiljanja.

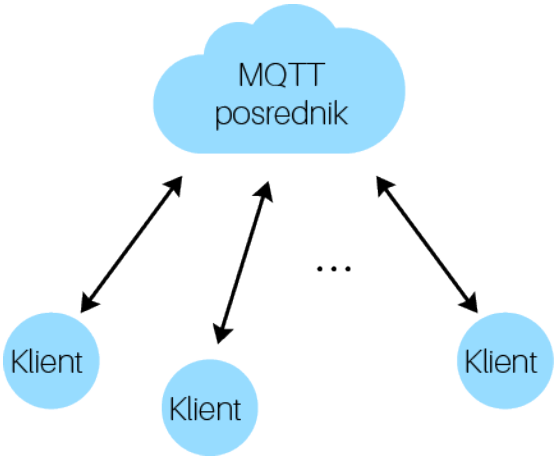
2.4.3 Uporabniška izkušnja

Ena glavnih metrik ocenjevanja kvalitete sporočilnega sistema je gotovo uporabniška izkušnja. Naš cilj je komunikacija v realnem času - to pomeni minimalno latenco, uporabnik nekega zamika ne sme zaznati. Nismo sicer zasledili, da obstaja kakšen standard oz. kakšna maksimalna latenca, ki je dovoljena za takšne storitve, opažamo pa da različni ponudniki takšnih storitev (npr. PubNub) zagotavljajo latenco pod 250 ms [12].

2.5 Poskusni scenarij

Ugotovili smo, da si bomo nalogo poenostavili, če uporabimo že implementirane MQTT kliente v kakšnem razširjenem programskem jeziku (izbrali smo Python) - napram klientu za uporabo iz ukazne vrstice so dosti bolj fleksibilni in nam omogočajo lažjo obdelavo podatkov.

Postavili smo prvi testni scenarij. Broker je tekel na oblaku, klienta pa na istem računalniku (broker/osebni računalnik). Slika 2.6 prikazuje komunikacijo. Klienta predstavljata publisherja in subscriberja, oba se povežeta na en kanal, prvi pošilja, drugi prejema. Vsako sekundo smo kot sporočilo pošiljali UNIX timestamp (32-bitno število). Prejemnik je pri sebi tako lahko izmeril čas, ki je bil potreben za obdelavo sporočila na brokerju.



Slika 2.6: Struktura našega scenarija.

2.6 Rezultati meritev

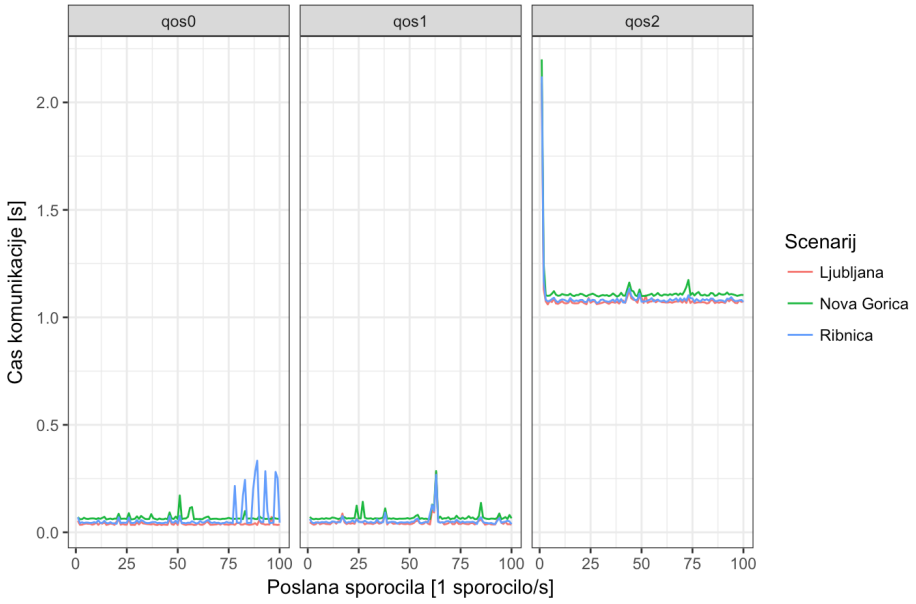
2.6.1 Čas

Lokalnost - eksperiment 1

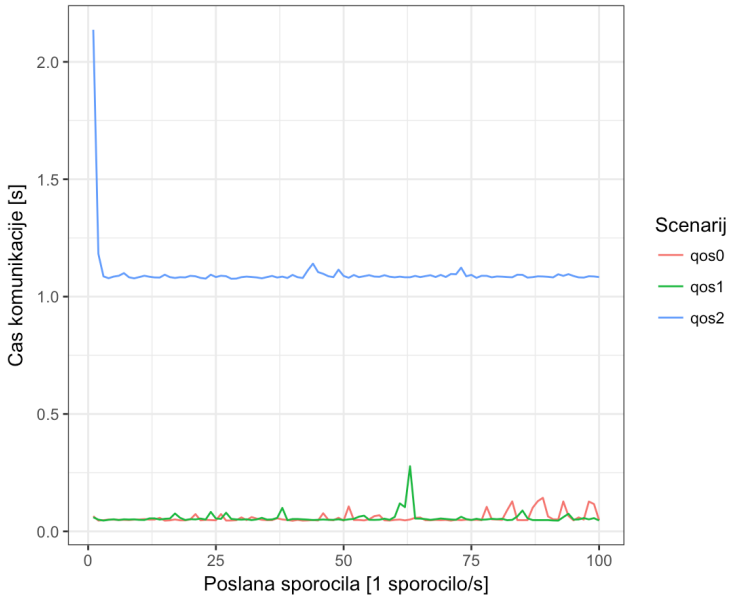
- **Hipoteza:** Zaradi oddaljenosti bodo največji časi pri pošiljanju v Novo Gorico. Z nastavitvijo $qos = 2$ bo pošiljanje trajalo najdlje.

- **Robni pogoji:** Eksperiment je bil izveden 9.4.2017 okoli 12:00, pošiljatelj je bil na računalniku v Kočevju, prejemniki v Ljubljani, Ribnici in Novi Gorici, posrednik pa je bil na oblaku DigitalOcean v Frankfurtu.

- **Komentar:** Iz slike 2.7 opazimo, da je bilo na lokaciji Nova Gorica pošiljanje res počasnejše. Prvo polovico naše hipoteze bi tako lahko potrdili, moramo pa se zavedati, da so bili časi odvisni tudi od omrežja. Na sliki 2.8, kjer je prikazan graf povprečja časov pošiljanja na vseh lokacijah, opazimo, da je za sporočila, poslana s parametrom *qos* nastavljenim na 2, pošiljanje najdaljše.



Slika 2.7: Graf časa komunikacije glede na lokacijo (Ljubljana, Ribnica, Nova Gorica).



Slika 2.8: Graf povprečnega časa pošiljanja sporočil na vseh lokacijah glede na različne nastavitve *qos*.

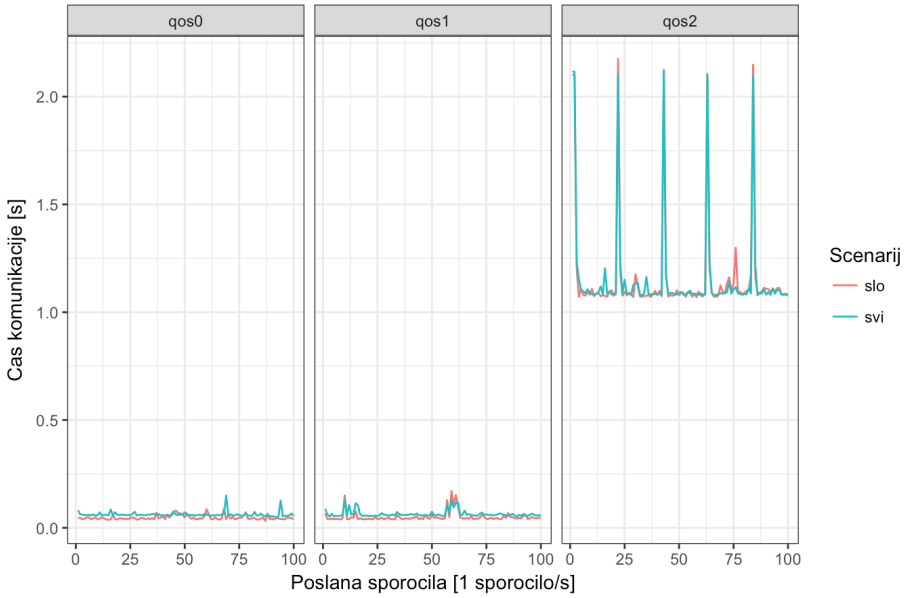
Lokalnost - eksperiment 2

Ker je bil eden naših članov ekipe na dopustu v Švici, smo se odločili, da ponovimo eksperiment lokalnosti in opazujemo, kako še večja oddaljenost vpliva na čase komunikacije.

- **Hipoteza:** Zaradi oddaljenosti bodo največji časi pri pošiljanju v Švico. Z nastavitvijo $qos = 2$ bo pošiljanje trajalo najdlje.
- **Robni pogoji:** Eksperiment je bil izveden 1.5.2017 okoli 10:50, pošiljatelj je bil na računalniku v Kočevju, prejemnika v Kočevju in v mestu Oberdiessbach v Švici (zračna razdalja med mesti pribl. 600km). Posrednik je bil na oblaku DigitalOcean v Frankfurtu.
- **Komentar:** Iz slike 2.9 lahko opazimo, kar smo pričakovali: čas komunikacije je za prejemnika v Švici malenkost daljši. To je zelo očitno pri nastavitvah $qos = 0$ in $qos = 1$, kjer je zeleno-modra črta konstantno nad oranžno. Pri $qos = 2$ pa razlika ni tako očitna, to bi lahko bilo zaradi rokovanja, ki se dogaja na posredniku.

Časi T

- **Hipoteza:** Z nastavitvijo $qos = 2$ bo pošiljanje trajalo najdlje. Lokalno pošiljanje bo potekalo hitreje.

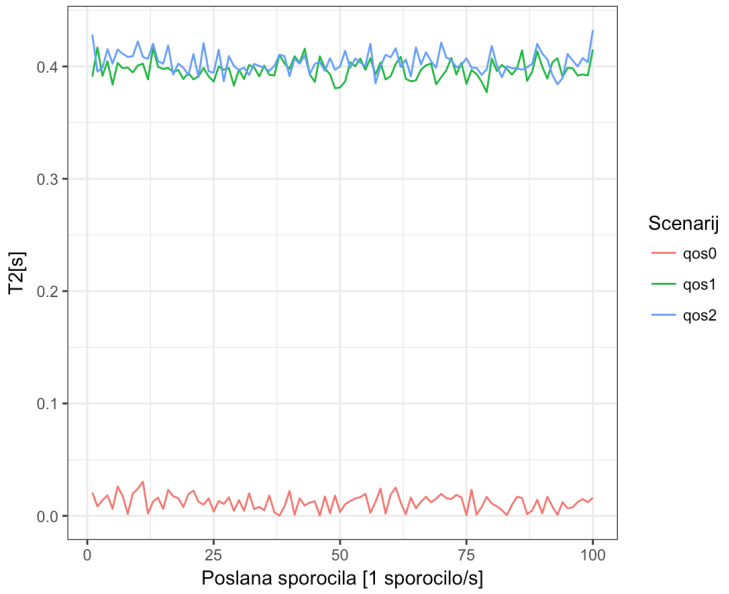


Slika 2.9: Graf časa komunikacije glede na lokacijo (Slovenija - Švica).

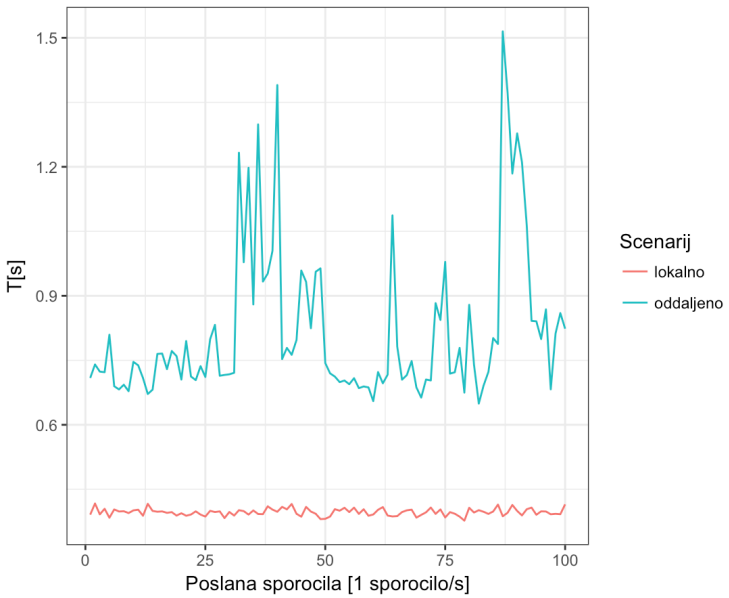
- **Robni pogoji:** Eksperiment je bil izveden 18.4.2017 okoli 19:30. Pri meritvi T_2 so bili klienti in posrednik na oblaku DigitalOcean v Frankfurtu. Za primerjavo meritev lokalno/nelokalno je potekala še komunikacija, ko sta bila klienta v Ljubljani, posrednik pa v Frankfurtu.

- **Komentar:** Slika 2.10 prikazuje graf časa T2 za vse 3 vrednosti parametra QoS, ko so tako klienti kot posrednik na oblaku. Na grafu ni videti nekih anomalij in pričakovano je, da ima $qos = 0$ najnižje vrednosti - v povprečju posrednik procesira 10 sporočil v 0.012 s, medtem ko za $qos = 1$ porabi 0.39 s, za $qos = 2$ pa 0.41 s. Presenetljivo je, kako blizu sta si časa, ko ima parameter *qos* vrednost 1 ali 2.
  Zanima nas seveda tudi primerjava obeh scenarijev - ko so klienti in posrednik skupaj v nekem lokalnem okolju ter ko so narazen. Ta prikaz imamo na sliki 2.11. Vrednost parametra *qos* je 1, klienta sta na računalniku v Ljubljani, posrednik pa je na oblaku v Nemčiji. Jasno je razviden vpliv omrežja, razlike v časih so večje, linija je bolj neenakomerna, kar pa je seveda pričakovano. Povprečje pošiljanja 10 sporočil v lokalnem okolju je bilo 0.39 s, v oddaljenem pa je 0.81 s. Zdaj lahko za ta primer izračunamo približek časov T_1 in T_3 (enačba 2.1).

$$T_1 = T_3 = \frac{T - T_2}{2} = 0.21s \tag{2.1}$$



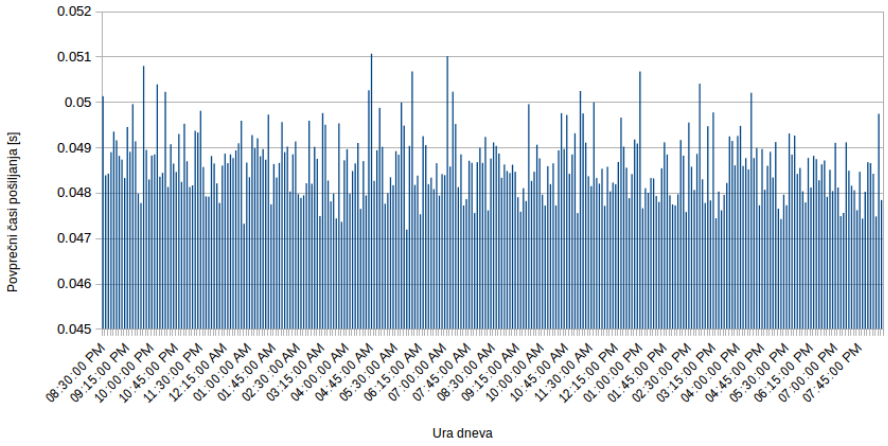
Slika 2.10: Graf časa T_2, ko so klienti in posrednik na oblaku.



Slika 2.11: Graf časa pošiljanja T, ko so klienti in posrednik na oblaku, $qos = 1$.

24 urni eksperiment

- **Hipoteza:** Preveriti želimo kako se časi pošiljanja sporočil spreminjajo čez dan v obdobju 24 ur. Sklepamo, da časi posiljanja nihajo glede na čas v dnevu.
- **Robni pogoji:** Eksperiment smo izvajali 24 ur, pognali smo ga 15.5.2017 ob 20:30. Na oblaku AWS smo poganjali skripto, ki je pošiljala in prejemala sporočila preko posrednika na oblaku Digital Ocean, skripta je vsakih 5 minut poslala 1000 sporočil na kanal. Simulirali smo 250 klientov, ki so prejemali sporočila. Odločili smo se za izvajanje skripte na oblaku, saj je tako lahko nemoteno delovala 24 ur. Oba oblaka Amazon AWS ter Digital Ocena sta locirana v Frankfurtu.
- **Komentar:** Na sliki 2.12 so prikazani povprečni časi prejemanja sporočil. V roku 24 ur ni opaziti nihanj v času pošiljanja. Časi so precej konstantni (okoli 0,05 sekunde), brez velikih odstopanj. Vzorca nihanja v 24 urah ne zaznamo.



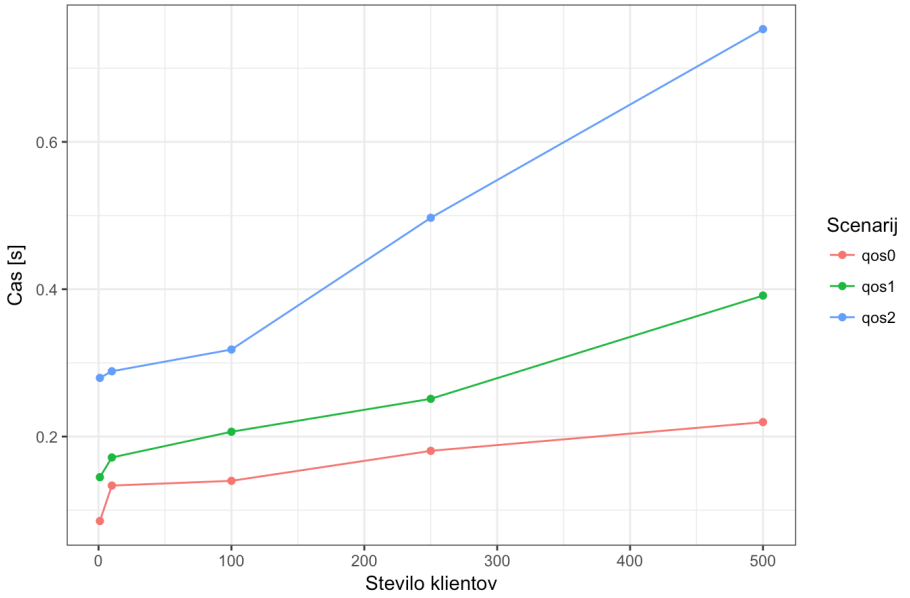
Slika 2.12: Graf časov pošiljanja sporočil v obdobju 24 ur.

2.6.2 Prepustnost

Odvisnost od števila klientov

- **Hipoteza:** Večje število klientov pomeni daljši povprečni čas komunikacije.
- **Robni pogoji:** Eksperiment je bil izveden 19.4.2017 okoli 21:00. Pri meritvi so bili klienti na računalniku v Ljubljani, posrednik pa na oblaku DigitalOcean v Frankfurtu.

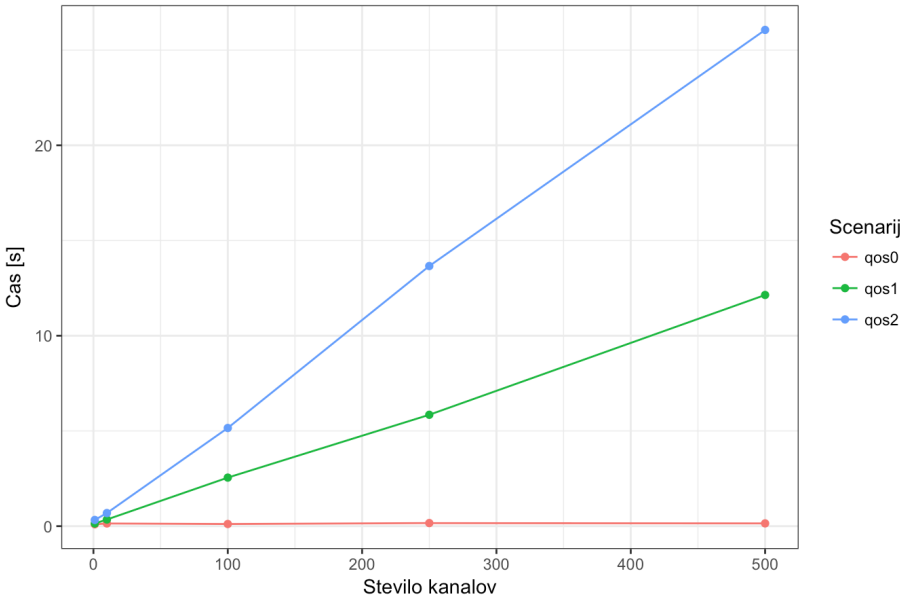
- **Komentar:** Iz slike 2.13 lahko potrdimo, da večje število klientov na istemu kanalu doprinese k daljšemu povprečnemu času dostave sporočil. Najbolj je to razvidno pri nastavitvi $qos = 2$, saj se daljša rokovanja med posrednikom in klienti seštejejo v večje pribitke časa potovanja sporočil. Vredno je tudi omeniti, da smo opazili tudi večja in pogostejša nihanja v času dostave sporočil, saj je posrednik zaradi večjega števila prijavljenih klientov bolj obremenjen.



Slika 2.13: Graf povprečnih časov dostave sporočil v odvisnosti od števila klientov.

Odvisnost od števila kanalov

- **Hipoteza:** Večje število kanalov pomeni daljši povprečni čas komunikacije.
- **Robni pogoji:** Eksperiment je bil izveden 19.4.2017 okoli 23:00. Pri meritvi so bili klienti na računalniku v Ljubljani, posrednik pa na oblaku DigitalOcean v Frankfurtu.
- **Komentar:** S povečevanjem števila kanalov se povprečni čas dostave sporočil povečuje, kar je jasno razvidno iz slike 2.14. Še posebej se dostavni čas poveča pri višji stopnji parametra *qos*, zaradi rokovanja med posrednikom in klienti.



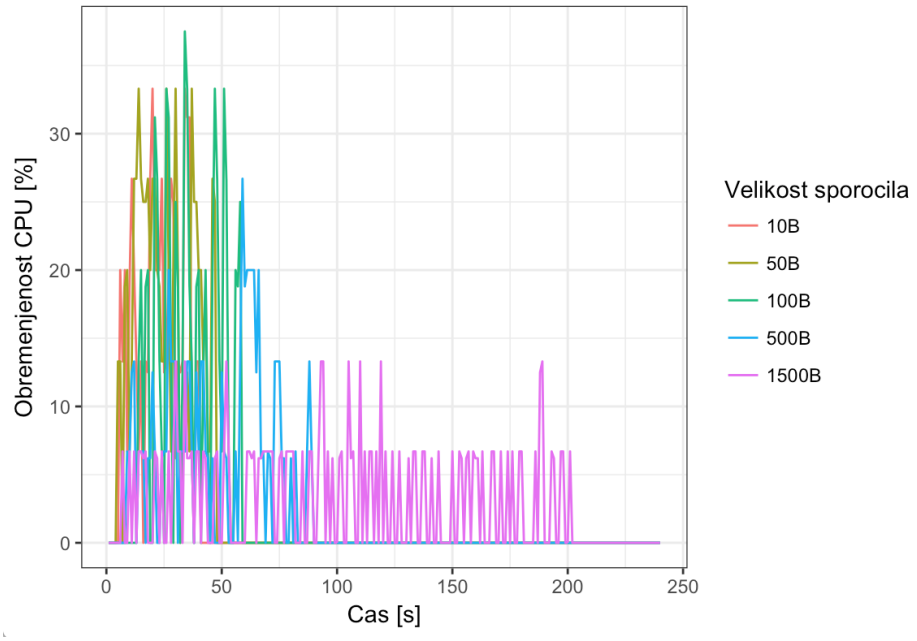
Slika 2.14: Graf povprečnih časov dostave sporočil v odvisnosti od števila kanalov.

Odvisnost od velikosti sporočil

- **Hipoteza:** Večja povprečna velikost sporočil pomeni daljši povprečni čas komunikacije in višjo porabo CPU na posredniku.

- **Robni pogoji:** Eksperiment je bil izveden 2.5.2017 okoli 18:00. Pri meritvi so bili tako klienti kot posrednik na oblaku DigitalOcean v Frankfurtu. Tokrat smo za obremenitev uporabili program Malaria [13] in ne svojih skript, saj je precej bolj zanesljiva in lažje konfigurabilna. Pri vseh eksperimentih je bilo simuliranih 10 klientov, vsak je pošiljal 10000 sporočil s frekvenco 1000/s in QoS = 0.

- **Komentar:** S povečevanjem velikosti sporočil se povprečni čas dostave sporočil povečuje, kar je razvidno iz slike 2.15 (prej se poraba CPU spusti na 0). Za porabo procesorja ne moremo reči enako, saj je pri dveh največjih velikostih poraba nižja. Opazimo tudi, da metoda merjenja porabe CPU (unixovski program *top*) ni čisto natančna, saj so vmes nihanja in tudi padci na 0. Porabo smo merili enkrat na sekundo. Za nižjo porabo CPU pri večjih sporočilih je mogoče razlog to, da posrednik neobdelana sporočila zaradi velikosti začasno shranjuje na disk (ker jih ne more vseh hraniti v RAMu) in obdeluje manj sporočil naenkrat. To se pozna tudi na prepustnosti, kar je razvidno iz tabele 2.1.



Slika 2.15: Graf porabe CPU na posredniku v odvisnosti od velikosti sporočil.

Tabela 2.1: Tabela prepustnosti in časa obdelave sporočil.

Velikost (B)	10	50	100	500	1500
Čas obdelave vseh sporočil (s)	49.16	55.19	63.24	108.24	239.45
Prepustnost (sporočila/s)	2649.73	2299.55	1936.44	1040.83	442.91

Uporabniška izkušnja

Če se držimo zgoraj omenjene meje 250 ms, lahko iz eksperimenta z več klienti (slika 2.13) opazimo, da pri nastavitvi $qos = 0$ naš sporočilni sistem zmore prenesti približno 500 klientov. Ker pa je vsak klient pošiljal sporočila s frekvenco 0,1 s in s tem dejansko simuliral več uporabnikov, lahko izračunamo, koliko uporabnikov je lahko aktivnih naenkrat. Recimo, da povprečen uporabnik v aktivnem pogovoru napiše 1 sporočilo na 5 sekund. To pomeni, da en klient simulira 50 uporabnikov, od koder sledi, da naš sporočilni sistem z enim posrednikom na oblaku Digital Ocean premore 25000 aktivnih uporabnikov, pri čemer je uporabniška izkušnja še vedno zadovoljiva.

2.6.3 Rezultati meritev v ekstremnih razmerah

Namen tega razdelka je izvesti meritve, ki bi obremenile posrednika, kolikor se le da.

Stresni test

V tej fazi smo želeli preizkusiti zmogljivost MQTT posrednika v našem oblaku in sicer, kako se obnese, ko je izpostavljen večjim bremenom. Za ta namen smo napisali dve skripti. Ena je ustvarila množico klientov, ki so se se prijavili na eno skupno temo, druga je ustvarila množico sporočevalcev, ki so to temo oblivali s sporočili. Test je potekal postopoma. Določili smo več različnih frekvenc, s katerimi so sporočevalci pošiljali sporočila. Pošiljanje sporočil z vsako frekvenco je potekali v nekem vnaprej določenem intervalu. To nam je omogočilo, da smo lahko opazovali, kako se obremenitev na strežniku s posrednikom spreminja v odnosu s frekvenco pošiljanja. Stanje na oblaku smo opazovali s pomočjo orodij oblačnega ponudnika in tudi direktno v virtualki.

- **Hipoteza:** Visoko obremenitev na oblaku bomo zaradi majhnih spročil težko dosegli. Ob povečevanju bremena se bo zvišala poraba CPE, poraba RAM pomnilnika pa bo nizka tudi ob večjih bremenih. Procesor bo predstavljal ozko grlo. Posrednika ne bomo obremenili do odpovedi.

- **Pogoji:** Za naše frekvence smo izbrali [5s, 1s, 0.5s, 0.1s, 0.01s]. Predstavljale naj bi nek realen čas pošiljanja pa tudi take čase, da močno obremenijo sistem. Naš interval pošiljanja je bil 70s na vsako frekvenco - torej skupen čas testa naj bi bil 350s, slabih 6min. Imeli smo 600 klientov naročnikov in 150 sporočevalcev. Testi so potekali 1x v večernih urah in 2x v jutranjih, kar pa sicer ni več tako pomembno, saj nas latenca tu ne zanima. Sporočila so bila nabor naključnih znakov, dolžine 300. Testirali smo 3x, 1x za vsak QOS. Na oblaku ob testiranju ni tekel noben drug proces, poraba RAM pomnilnika je bila okoli 17%, CPE pa okoli 0,6%.

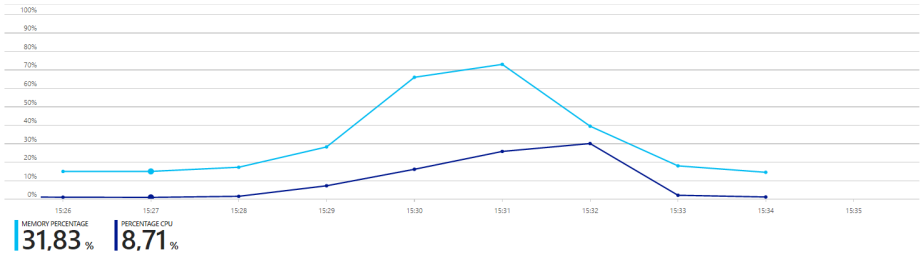
- **Rezultati:**

  - QOS=0

    V nasprotuju s pričakovanji vidimo na sliki 2.16, da v tem primeru močno naraste uporaba RAM pomnilnika in manj poraba CPE. Graf nam lepo prikaže povečevanje potrebo po resursih, bolj ko se povečujejo frekvence pošiljanja. Najbolj interesantno pa je, da se je ob največi frekvenci pošiljanja na 0.01s, posrednik sesul. Posrednik je ob večjih frekvencah začel zavračati sporčevalce in odklapljal naročnike. Ob sesutju je prekinil vse povezave s sporočevalci in naročenimi klienti.

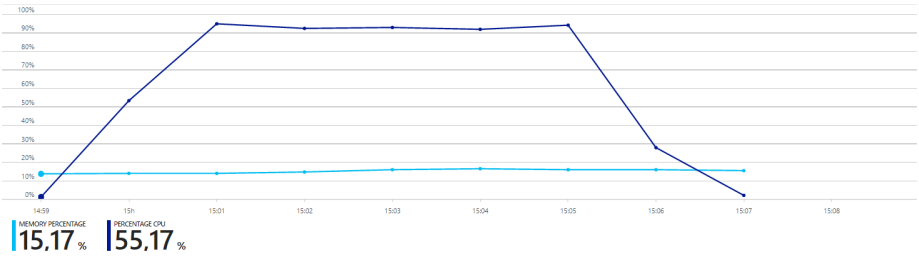
  - QOS=1

    Pri QOS=1 je poraba RAM pomnilnika minimalna, poveča se obremenitev CPE (Slika 2.17). Ob pošiljanju sporočil, ki so manj kot 1s



Slika 2.16: Graf obremenitve sistema, qos=0.

narazen, se poraba CPE dvigne na skoraj 100%. Začenjajo se čakalne vrste sporočil, saj jih posrednik hitreje dobiva, kot jih lahko razpošilja. Prišlo je do nekaj odklapljanj klientov, sporočevalcev, vendar nič hujšega in sporočanje je bilo zanesljivo, še bolj pomembno pa, da se MQTT posrednik ni sesul.

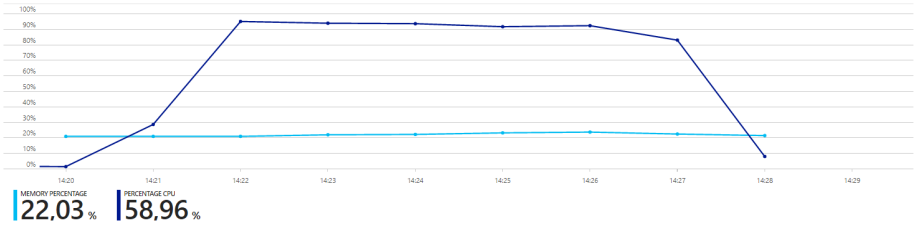


Slika 2.17: Graf obremenitve sistema, qos=1.

- QOS=2

  Zgodba je podobna kot pri QOS=1. Se pa pri večjih frekvencah pošiljanja že močneje kopiči čakalna vrsta soročil, saj vsako potrebuije še več časa, da zanesljivo pride do klientov. Na sliki 2.18 ta pojav lahko opazimo proti koncu testa, ko se obremenjenost CPE že počasi zmanjšuje. Na tej točki so nekateri sporočevalci že zaključili s pošiljanjem, drugi, ki so čakali pa še ne in tako je CPE manj obremenjen. Zato je tudi test trajal dlje - okoli 8min. Tudi RAM poraba je v primerjavi s QOS=1 večja. Znova je komunikacija zanesljiva, čeprev počasnejša.

- **Komentar:** V nasprotju s hipotezo smo posrednika lahko močno obremenili in celo sesuli. Pri QOS=1/2 izgleda da je, kot smo predvidevali, ozko grlo CPE, RAM se pravzaprav ni veliko bremenil. Pri QOS=0 pa izgleda obratno, saj je poraba delovnega pomnilnika zelo velika. Ker se je storitev sesula, slutnjo težje potrdimo.



Slika 2.18: Graf obremenitve sistema, qos=2.

Obremenjevanje posrednika s hranjenjem sporočil

Cilj testa je, da bi videli, koliko sporočil lahko shrani posrednik, če sporočila samo prejema in jih ne pošilja naprej. Ko je dosežen nek limit, vklopimo prejemnike in opazujemo porabo procesorja na posredniku. Hranili bi sporočila s QoS = 1 ali 2. Pošiljali bi veliko sporočil na veliko topicov in opazovali, koliko jih je sposoben hraniti.

- **Hipoteza:** Posrednik bo pri določeni količini (predvidoma bo le-ta visoka) sporočil zapolnil celoten razpoložljiv pomnilnik in ne bo hotel sprejemati novih sporočil / se bo sesul. če bomo prejemnike vklopili pred tem, bo poraba CPU zelo visoka in časi dostave sporočil bodo dolgi.

- **Pogoji:** Pri tem testu smo uporabili sicer ne najbolj smiselno konfiguracijo brokerja: shranjevanje sporočil na disk (*persistence*) smo izklopili - posledično posrednik sporočila hrani samo v RAMu. Na virtualki smo imeli na voljo 1GB RAMa in 30GB diska, zato je bolj smiselno, da poskušamo zapolniti RAM. Poleg tega smo nastavili maksimalno število sporočil, ki jih hrani, na neskončno (torej je omejitev le kapaciteta RAMa).

  Testirali smo dva različna scenarija: obremenjevanje do konca in pravočasni vklop prejemnikov.

  Pri pravočasnem vklopu prejemnikov smo imeli 1000 topicov, 10 pošiljateljev - vsak je poslal 10000 sporočil z 20 znaki (interval med sporočili je bil 0.0001s) in 10 prejemnikov. Tako pošiljatelji kot prejemniki so bili enakomerno porazdeljeni po topicih. Skripta, ki je simulirala pošiljatelje, je tekla na isti virtualki kot posrednik, skripta, ki je simulirala prejemnike, pa lokalno na našem računalniku v Ljubljani.

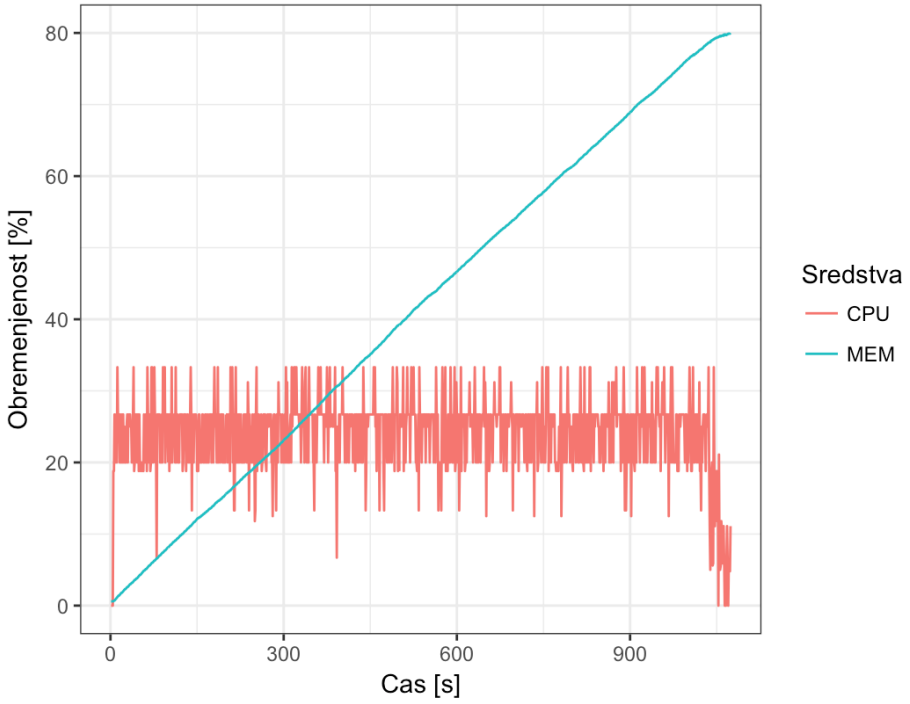
  Pri obremenjevanju do konca je bila situacija skoraj enaka, le da pošiljateljev nismo omejili s številom sporočil.

- **Rezultati:**

  - Obremenjevanje do konca:

    Tu so rezultati pričakovani: poraba RAMa počasi, linearno narašča, dokler ne doseže meje - to vidimo na sliki 2.19. Pri nas je bila ta

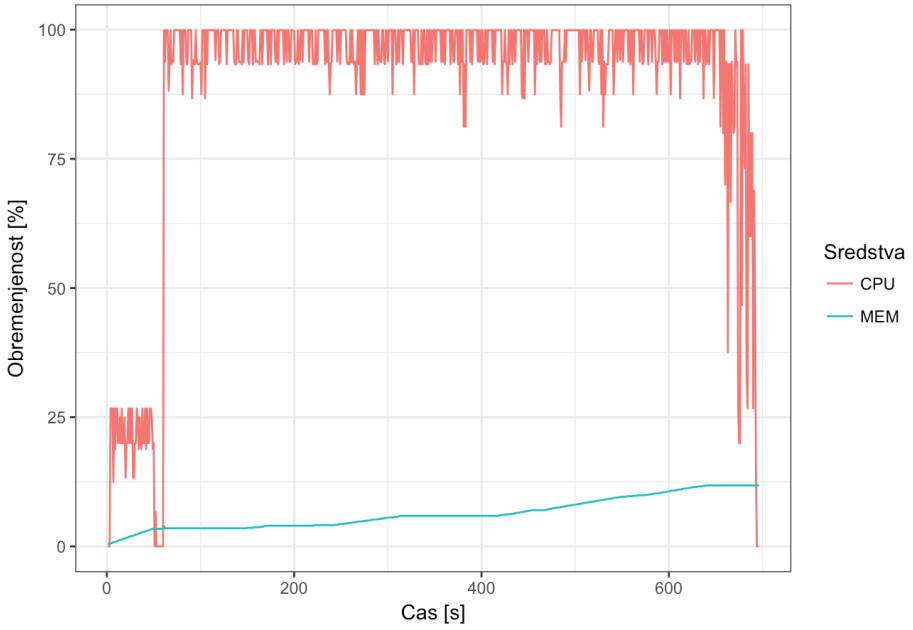
meja približno 80%. Pri tej meji je sistem postal neodziven, ukazov v ukazni vrstici nismo mogli izvajati in sistem je začel prikazovati opozorila. To se je zgodilo po približno 2700000 poslanih sporočilih. Poraba CPU je rahlo nihala, vendar je v mejah normale glede na število prejetih sporočil.



Slika 2.19: Graf obremenitve sistema pri obremenjevanju do konca.

– Pravočasni vklop prejemnikov:

  Rezultat potrjuje našo hipotezo: ob nenadnem vklopu prejemnikov je poraba CPU sunkovito narasla do maksimuma in tam ostala, dokler prejemnik ni razposlal vseh sporočil. Razpošiljanje 100000 sporočil je trajalo dobrih 10 minut (636 sekund). Na sliki 2.20 se to lepo vidi. Prvih 50 sekund je potekalo pošiljanje sporočil in tu je bila poraba CPU nižja. Poraba pomnilnika je počasi naraščala (smiselno, kajti sporočila je posrednik shranjeval v RAM). Nenavadno se nam zdi, da je tudi med razpošiljanjem naraščala. Morala bi ostati enaka, kajti posrednik tedaj ni več prejemal sporočil (znižala se seveda ni, ker smo nastavili, da posrednik vsa sporočila s QoS $>= 1$ shranjuje). To anomalijo bi pripisali usmerjanju sporočil znotraj posrednika, ki pri tem ni najbolj efektiven.



Slika 2.20: Graf obremenitve sistema pri vklopu prejemnikov.

2.7 Zaključek

Naš cilj je bil preveriti zmogljivost in zanesljivost sporočilnega sistema, ki temelji na protokolu MQTT. Uporabili smo odprtokodni posrednik Mosquitto, ki je tekel na oblaku DigitalOcean. Izvajali smo teste, kjer smo poskušali identificirati vpliv različnih spremenljivk na zmogljivost našega sistema. Omejili smo se na lokalnost, število klientov, frekvenco pošiljanja sporočil, število kanalov, velikost sporočil, sistem pa smo poskusili na različne načine tudi maksimalno obremeniti s ciljem, da najdemo ozka grla.

Izkazalo se je, da je glavno ozko grlo sistema CPU, pri posebnih nastavitvah (ki jih v praksi sicer ne bi uporabili) pa lahko predstavlja problem tudi kapaciteta delovnega spomina. Na porabo CPU najbolj vpliva število sporočil, ki jih mora posrednik obdelati. Velikost sporočila naj v nekem realnem scenariju ne bi predstavljala težav, saj je dolžina povprečnega sporočila, ki ga ljudje pošljemo preko takih sistemov, pod 100 B.

Naše ugotovitve smo primerjali s sorodnimi članki, ki so prišli do podobnih ugotovitev. Avtorji članka [14], ki raziskuje vpliv QoS na latenco pošiljanja, so prišli do enake ugotovitve kot mi - večji QoS pomeni večjo latenco. V članku [15] smo dobili vpogled v zmogljivost protokola MQTT v primerjavi s sorodnimi protokoli, primerjali so tudi različna posrednika za protokol MQTT. Izkaže se, da

je zaprtokodni plačljivi posrednik HiveMQ veliko bolj zmogljiv kot Mosquitto, kar ni precej presenetljivo.

Na koncu lahko sklenemo, da je protokol MQTT in implementacija posrednika Mosquitto zelo sposoben in optimiziran sistem že za delovanje na manj zmogljivih sistemih. Posrednik je imel na oblaku namreč le 1 jedro CPU in 1 GB delovnega pomnilnika, česar dandanes v produkciji skoraj več ne srečamo. Z veliko gotovostjo lahko torej trdimo, da bi se sistem v praksi dobro obnesel tudi pod velikim bremenom in je primeren za uporabo v oblačnih storitvah, kot je naša.

Poglavje 3

Analiza zmogljivosti spletne aplikacije Fear Index

Dejan Dolenc, Marko Krajinović, Kristjan Panjan

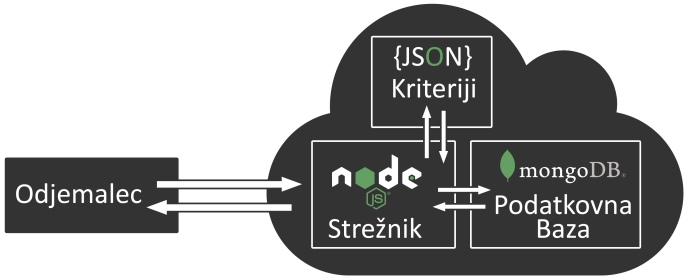
3.1 Opis problema

V okviru projekta Think Again smo razvili spletno aplikacijo Fear Index za merjenje stopnje ustrahovanja v spletnih člankih in poročilih. Aplikacija sprejme besedilo, ga pregleda in izračuna stopnjo ustrahovanja, ki jo ob branju povzroča članek. Programska koda je napisana na osnovi modela Node.js in MongoDB. Večina oblačnih storitev ne podpira uporabe teh dveh razvojnih okolij, zato je potrebno izbrati ustreznega ponudnika. Zanima nas, koliko uporabnikov istočasno lahko nemoteno uporablja našo storitev in kako dolga besedila še lahko obdela naš program.

Slika 3.1 prikazuje shemo oblačnega sistema. V oblaku teče strežnik s spletno aplikacijo, ki je povezan s podatkovno bazo člankov in seznamom kriterijev za analizo besedila. V oblak je povezan odjemalec, ki strežniku pošilja zahtevke.

3.2 Namen

Cilj pričujočega poglavja je analizirati zmogljivost spletne aplikacije na podlagi različnih metrik. Z vnaprej pripravljenimi testnimi primeri bomo izmerili



Slika 3.1: Shema oblačnega testnega sistema.

skrajne zmožnosti naše aplikacije in rezultate primerjali z referenčnim bremenom.

Na osnovi meritev bomo lahko natančno ocenili kolikšno število zahtev storitev še lahko obdela in kakšna je meja dolžine besedila, ki ga lahko analiziramo.

3.3 Izbira oblačnih ponudnikov

V pričujočem razdelku pregledamo razpoložljivo ponudbo za gostovanje naše storitve v oblaku. Večina ponudnikov ne podpira uporabe tehnologij potrebnih za tek naše aplikacije, zato smo pregledali pakete ponudnikov, ki nudijo najem virtualk. Na virtualko lahko namestimo vsa potrebna orodja in jo prilagodimo za uporabo storitve.

3.3.1 Primerjava ponudnikov

V tabeli 3.1 na strani 39 so prikazani podatki o ponudnikih oblačnih storitev.

Najcenejši ustrezni paket ponuja Dedimax [16], ki stane 1,90€ mesečno. Gre za osnovni paket in vsebuje virtualen stroj, ki mu pripada le enojedrni procesor, 1GB delovnega pomilnika in 10GB trdega diska. Podatek o taktu ure procesorja na ponudnikovi spletni strani ni bil naveden. Za našo storitev ima dovolj trajnega in delovnega pomnilnika.

Cenovno je najbolj učinkovit paket ponudnika Contabo [20], ki stane 6,99€ mesečno. Ta vsebuje dovjedrni procesor, 6GB delovnega pomnilnika in 500GB trdega diska.

Ponudnik	Cena[€]	CPU	RAM[GB]	Disk[GB]	Tip pondudbe
Dedimax [16]	1,90	1 jedro	1	10	virtualka
1and1 [17]	4,99	n/a	0,5	30	virtualka
DigitalOcean [18]	5,00	1 jedro	0,5	20	virtualka
King-Servers [19]	5,00	n/a	0,5	15	virtualka
Contabo [20]	6,99	2 jedri	6	500	virtualka
Dedimax [16]	6,99	n/a	2	160	virtualka
GoDaddy [21]	14,99	n/a	1	40	virtualka
Hostgator [22]	19,95	2 jedri	2	120	virtualka
Bluehost [23]	19,99	2 jedri	2	30	virtualka
vps.net [24]	24,99	n/a	4	75	virtualka

Tabela 3.1: Primerjava specifikacij paketov oblačnih ponudnikov.

3.3.2 Strežnik v oblaku

Za testiranje storitve v oblaku, smo izbrali paket ponudnika Dedimax, ki stane 1,90€. Zanima nas ali za zadovoljivo delovanje naše storitve zadošča najcenejši primerni paket. Po zakupu strežnika smo preverili še njegove natančne specifikacije, ki niso bile navedene na spletni strani. Takt ure procesorja je 3,70GHz.

3.3.3 Domači strežnik

Storitveno aplikacijo lahko namestimo tudi na domači strežnik. Namesto, da najamemo virtualko v oblaku, uporabimo prost domači računalnik in nanj namestimo Linux strežnik. Glavna prednost tega pristopa je, da imamo vse potrebno že na voljo in nam ni treba plačevati mesečne najemnine za storitve gostovanja. Poleg tega lahko vse namestimo in urejamo neposredno na strojni opremi. Glavna slabost pri tem pristopu je vzdrževanje strežnika. Če katera izmed komponent neha delovati, jo moramo sami zamenjati. Takrat moramo sistem namestiti nekje drugje. Če tega ne storimo, storitev ne bo na voljo. Prav tako veliko težje razširimo sistem, če potrebujemo dodatne resurse. Uporabljali smo računalnik z dvojedrnim procesorjem s taktom ure 3,07GHz, 4GB delovnega pomnilnika in 500GB trdega diska. Primerljivo virtualko s seznama v tabeli ponuja Contabo [20] za 6,99€ mesečno. Spletno aplikacijo bomo namestili na oba strežnika in primerjali rezultate.

3.4 Izbira tehnologij

V tem razdelku so na kratko opisane vse izbrane tehnologije, ki smo jih uporabili pri razvoju aplikacije in tiste, ki jih potrebujemo za analizo zmogljivosti.

3.4.1 Node.js

Pri razvoju aplikacije smo uporabili Node.js [25]. To je odprtokodno izvajalno okolje, ki deluje na osnovi javascript-a. Uporablja dogodkovni model in je zelo

učinkovito. Prav tako je zanj na voljo veliko odprtokodnih knjižnic, ki olajšajo pisanje programov. V okolju Node.js smo napisali glavni algoritem za analizo besedila.

3.4.2 MongoDB

Seznam in vsebine privzetih člankov hranimo v podatkovni bazi zgrajeni s programom MongoDB [26]. To je odprtokodni program za podatkovne baze. Podatke hrani in organizira v dokumentih, ki imajo strukturo podobno JSON standardu.

3.4.3 Strežnik

Aplikacija teče na domačem strežniku z operacijskim sistemom Ubuntu [27]. To je ena izmed najbolj podprtih različic sistema Linux. Strežnik je povezan v lokalno omrežje in internet. Nanj se povežemo z lokalnim in oddaljenim odjemalcem od koder izvajamo teste zmogljivosti z orodjem za testiranje.

3.4.4 Odjemalec

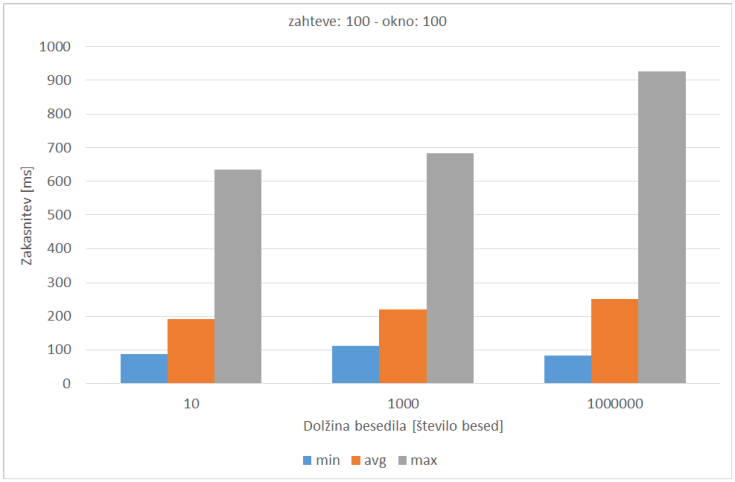
Aplikacijo testiramo s pametnim odjemalcem, ki smo ga napisali z Node.js. Odjemalec omogoča testiranje storitve z različno dolgimi tekstovnimi datotekami. Izbrana besedila pošilja strežniku in meri čas med oddajo zahteve in prejemom strežnikovega odgovora. Po končanem merjenju vrne čas v milisekundah. Odjemalec omogoča merjenje trajanja obdelave enega zahtevka ter povprečnega ali skupnega trajanja več zaporednih zahtevkov.

3.5 Bremenske datoteke

Spletna aplikacija Fear Index je namenjena za ocenjevanje stopnje ustrahovanja v člankih in drugih medijskih besedilih. Teksti, ki jih bodo aplikaciji pošiljali uporabniki, imajo določeno predvideno dolžino. Pregledali smo dolžine 97 besedil iz podatkovne baze člankov ter izračunali povprečje. Te segajo od 80 besed do malo več kot 2000. Povprečna dolžina vseh izmerjenih znaša 865 besed. Besedila take dolžine so naše referenčno breme, ki predstavlja realne datoteke uporabnikov.

3.6 Strategija okna

Okno določa število zahtevkov, ki jih lahko vzporedno pošljemo. Ko se okno napolni, prenehamo pošiljati, dokler ne dobimo odgovora na enega izmed zahtevkov. Po prejetem odgovoru pošljemo nov zahtevek, da ponovno napolnimo okno. Uporaba okna nam omogoča, da v vsakem trenutku kontroliramo koliko



Slika 3.2: Graf zakasnitev pri 100 sočasnih zahtevah z različno dolgimi datotekami. Lokacija: Ljubljana; server: domači.

zahtevkov je na strežniku za obdelavo. Na ta način hitro obremenimo strežnik, hkrati pa simuliramo omejeno množico uporabnikov, ki sočasno uporablja storitev.

3.7 Meritve zakasnitev domačega strežnika

V pričujočem razdelku so prikazane meritve testov zakasnitev do domačega strežnika v Škofji Loki. Merili smo iz dveh različnih lokacij, da analiziramo razlike v zakasnitvah.

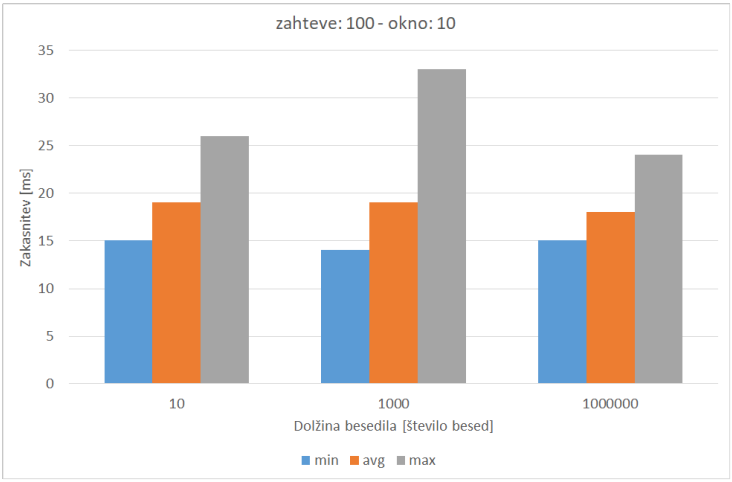
3.7.1 Meritve iz Ljubljane

Grafi na slikah od 3.2 do 3.5 prikazujejo meritve iz Ljubljane. Merili smo minimalne, povprečne in maksimalne zakasnitve pri različnem številu zahtev z različno velikim oknom in pri različnih dolžinah besedila.

Merili smo zakasnitve pri 100 zahtevah. Najprej smo sprožili vseh 100 hkrati, nato smo okno zmanjšali na 10.

Iz slike 3.2 lahko razberemo, da se minimalna zakasnitev spreminja neodvisno od števila besed v testni datoteki. Povprečna zakasnitev raste linearno, maksimalna pa drastično zraste pri preskoku iz tisoč na milijon besed.

Na osnovi rezultatov iz slike 3.3 sklepamo, da dolžina besedila, ko okno omejimo na 10, ne vpliva na zakasnitev. Maksimalna zakasnitev nima jasne



Slika 3.3: Graf zakasnitev 100 zahtev pri oknu velikem 10 z različno dolgimi datotekami. Lokacija: Ljubljana; server: domači.

korelacije z dolžino datoteke. Sklepamo, da na različne zakasnitve vpliva obremenitev omrežja ob času testiranja.

Isti test smo ponovili še pri 500 zahtevah.

Ko primerjamo sliko 3.4 s sliko 3.2, opazimo da večje število zahtev močno vpliva na zakasnitve. Ko smo povečali zahteve s 100 na 500 so zakasnitve zrastle več kot desetkrat.

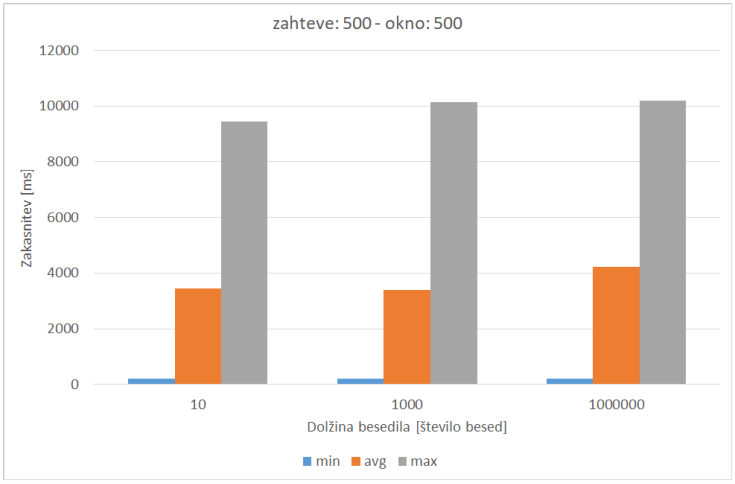
Iz slik 3.3 in 3.5 je razvidno, da večje število skupnih zahtev ne vpliva močno na zakasnitve, če velikost okna ostane enaka.

3.7.2 Meritve iz Bele Krajine

Meritve pri 100 zahtevah smo ponovili še iz Bele Krajine.

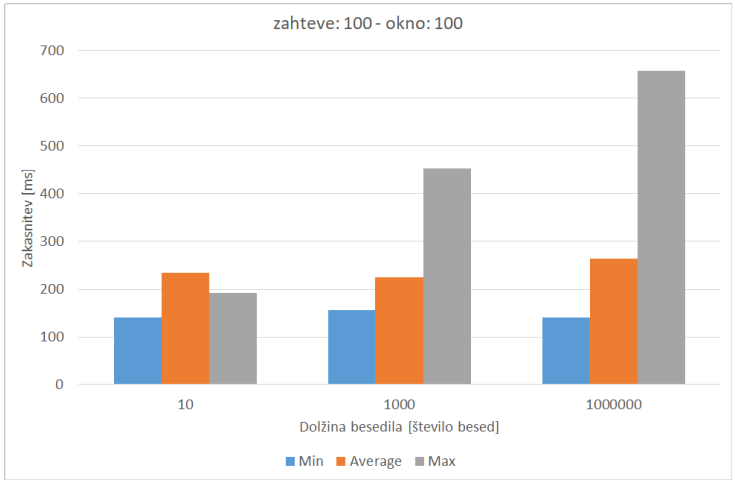
Sliki 3.2 in 3.6 prikazujeta enake meritve vendar iz dveh različnih lokacij. Različne zakasnitve omrežja so glavni razlog za razlike v rezultatih.

Podobne rezultate vidimo tudi ob primerjavi slik 3.3 in 3.7. Meritve iz Bele Krajine imajo večje zakasnitve. To lahko razložimo s primerjavo omrežnih zakasnitev do strežnika. Skupna zakasnitev do strežnika iz Ljubljane je veliko manjša kot iz Bele Krajine. To je razvidno iz meritev na slikah 3.8 in 3.9.

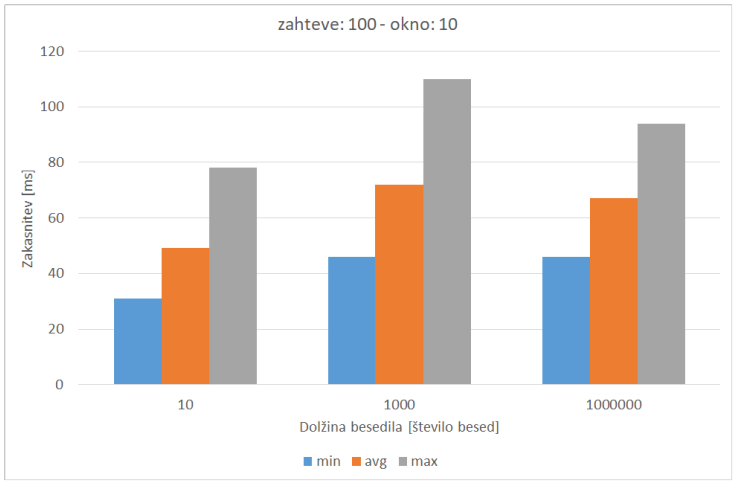


Slika 3.4: Graf zakasnitev pri 500 sočasnih zahtevah z različno dolgimi datotekami. Lokacija: Ljubljana; server: domači.

Slika 3.5: Graf zakasnitev 500 zahtev pri oknu velikem 10 z različno dolgimi datotekami. Lokacija: Ljubljana; server: domači.



Slika 3.6: Graf zakasnitev pri 100 sočasnih zahtevah z različno dolgimi datotekami. Lokacija: Bela Krajina; server: domači.



Slika 3.7: Graf zakasnitev 100 zahtev pri oknu velikem 10 z različno dolgimi datotekami. Lokacija: Bela Krajina; server: domači.

```
Tracing route to dquad.ddns.net [89.142.51.109]
over a maximum of 30 hops:

  1     2 ms    <1 ms    <1 ms  88.200.102.1
  2    <1 ms     3 ms     3 ms  192.168.200.33
  3     2 ms     2 ms     2 ms  lrozna-1-v114.sd-lj.si [212.235.230.109]
  4    11 ms    11 ms     2 ms  lljtpl2-v822.arnes.si [178.172.80.98]
  5    42 ms     1 ms    <1 ms  lljtpl1-v484.arnes.si [88.200.2.46]
  6    <1 ms    <1 ms    <1 ms  larnes6-v400.arnes.si [88.200.7.237]
  7     3 ms     3 ms     3 ms  six.siol.net [91.220.194.105]
  8     3 ms     3 ms     3 ms  95.176.255.240
  9     5 ms     5 ms     5 ms  BSN-142-51-109.dynamic.siol.net [89.142.51.109]

Trace complete.
```

Slika 3.8: Pot in omrežne zakasnitve do strežnika iz Ljubljane; server: domači.

```
Tracing route to dquad.ddns.net [89.142.51.109]
over a maximum of 30 hops:

  1    13 ms    <1 ms    <1 ms  192.168.64.1
  2    24 ms    17 ms    21 ms  89-212-0-1.gw.t-2.net [89.212.0.1]
  3    23 ms    13 ms    32 ms  84.255.208.129
  4    24 ms    10 ms     9 ms  84-255-250-217.core.t-2.net [84.255.250.217]
  5    39 ms    11 ms    37 ms  six.siol.net [91.220.194.105]
  6    16 ms    28 ms    21 ms  95.176.255.240
  7    30 ms    28 ms    21 ms  BSN-142-51-109.dynamic.siol.net [89.142.51.109]
```

Slika 3.9: Pot in omrežne zakasnitve do strežnika iz Bele Krajine; server: domači.

3.7.3 Meritve zakasnitev pri različnih velikostih okna

Merili smo tudi zakasnitve pri različno velikih oknih in enaki dolžini besedila. Te meritve smo opravljali v Ljubljani.

Iz slike 3.10 je razvidno, da pri večanju števila hkratnih zahtev močno vplivamo na zakasnitve.

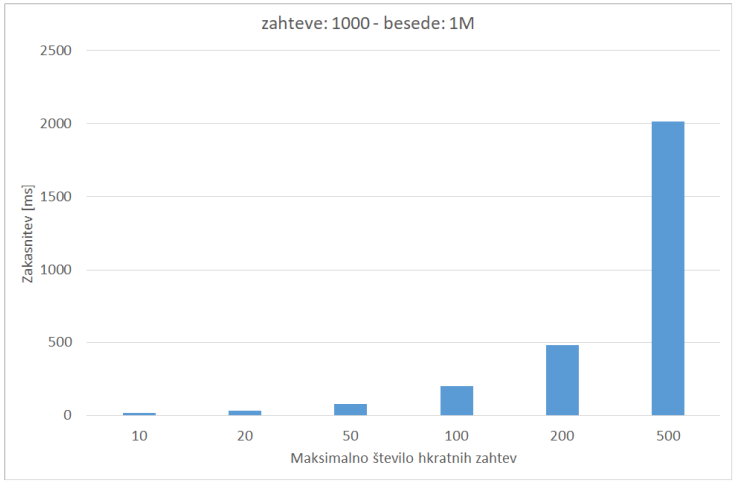
Ko primerjamo sliki 3.10 in 3.11, opazimo, da dolžina besedila minimalno vpliva na razliko v zakasnitvah. Sklepamo, da sprejemanje zahtev in pošiljanje odgovorov obremeni strežnik mnogo bolj kot analiza besedila.

3.8 Meritve zakasnitev strežnika v oblaku

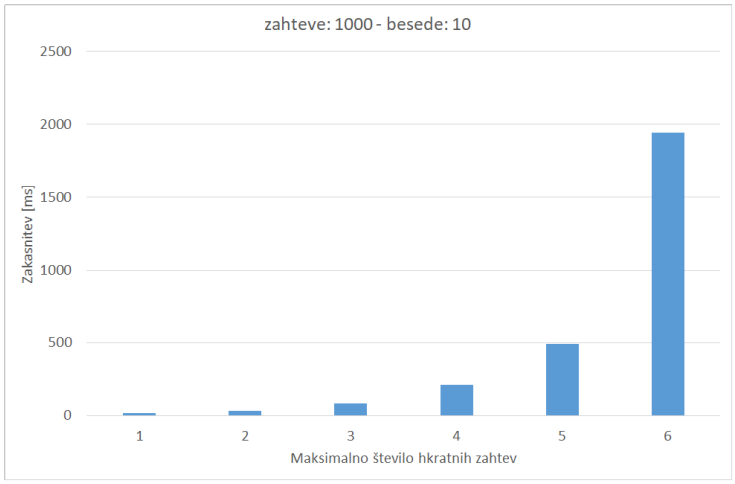
Teste smo ponovili tudi na oblačnem strežniku in primerjali izmerjene rezultate.

Ko primerjamo sliki 3.2 in 3.12 opazimo, da so zakasnitve domačega strežnika dvakrat višje, maksimalne pa še drastično višje.

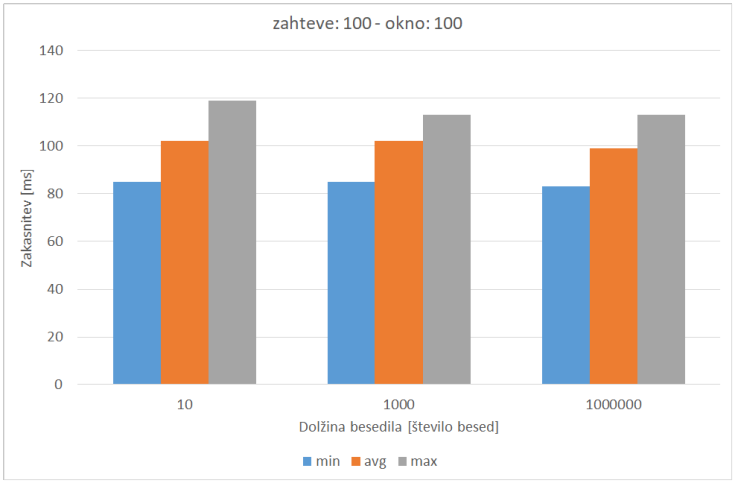
Ko okno omejimo na 10, se zakasnitve znižajo. Povprečne zakasnitve padejo s 100 *ms* na 65 *ms* (sliki 3.12 in 3.13). Pri domačem strežniku je ta sprememba veliko večja. V tem primeru je sprememba z 200 *ms* na 18 *ms* (sliki 3.2 in 3.3).



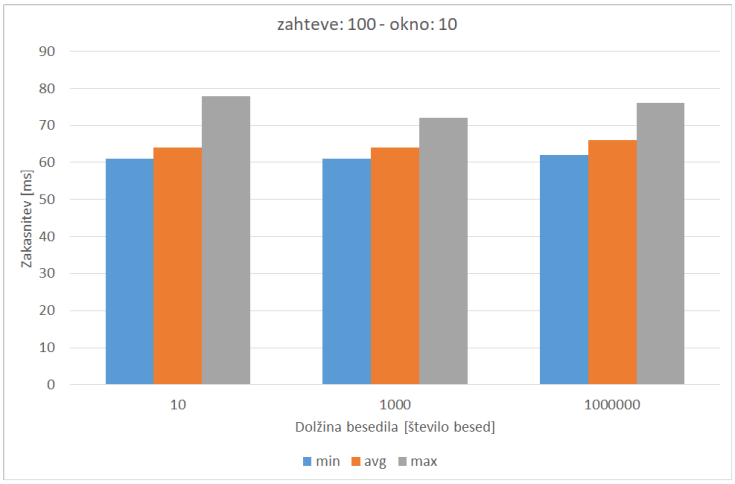
Slika 3.10: Graf zakasnitev pri večanju hkratnega števila zahtev. Dolžina besedila je milijon besed. Lokacija: Ljubljana.



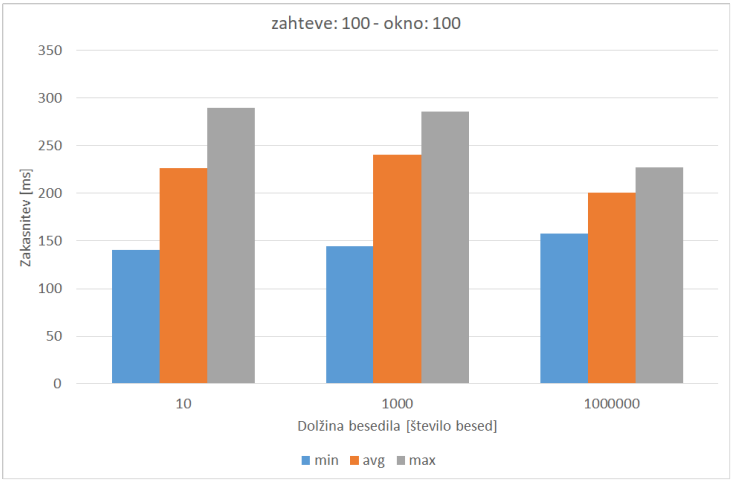
Slika 3.11: Graf zakasnitev pri večanju hkratnega števila zahtev. Dolžina besedila je deset besed. Lokacija: Ljubljana.



Slika 3.12: Graf zakasnitev pri 100 sočasnih zahtevah z različno dolgimi datotekami. Lokacija: Ljubljana; server: cloud.



Slika 3.13: Graf zakasnitev 100 zahtev pri oknu velikem 10 z različno dolgimi datotekami. Lokacija: Ljubljana; server: cloud.



Slika 3.14: Graf zakasnitev pri 100 sočasnih zahtevah z različno dolgimi datotekami. Lokacija: Bela Krajina; server: cloud.

Rezultati testov iz Bele Krajine imajo ponovno večjo zakasnitev zaradi zakasnitev omrežja. To lahko razberemo iz slik 3.12 in 3.14. Povprečne zakasnitve se povečajo s faktorjem, ki je večji od 2. Razlika med njimi je 125 *ms*.

Če vseh 100 zahtevkov sprožimo hkrati na domači strežnik in strežnik v oblaku, ne vidimo bistvene spremembe (sliki 3.6 in 3.14). Ko okno omejimo na 10 opazimo, da so zakasnitve oblačnega strežnika približnio 40*ms* krajše (grafa 3.7 in 3.15).

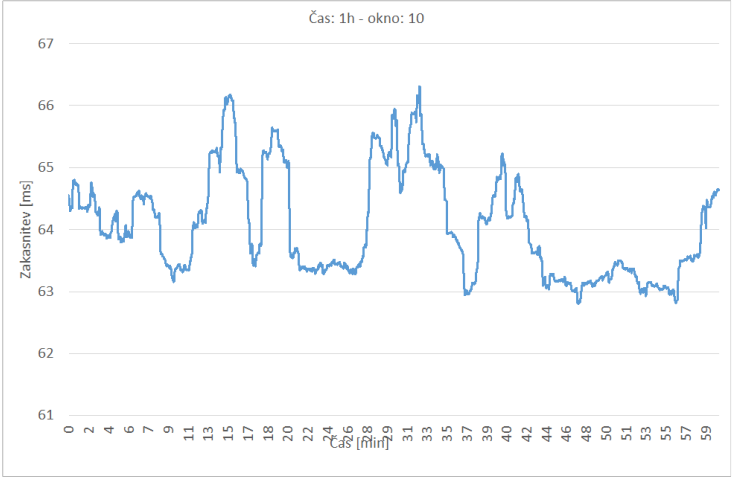
3.9 Testi daljše obremenitve

Zanimalo nas je, kako se strežnik odziva pri neprestani uporabi več uporabnikov, ki traja dlje časa. Teste smo opravili z datotekami realne dolžine. Tokrat smo teste omejili časovno namesto s številom zahtevkov. Hkrati smo tudi merili porabo resursov na strežniku, da lahko ocenimo koliko lahko še dodatno obremenimo strežnik. Testirali smo eno uro pri treh različnih oknih: 10, 100 in 1000. Nato smo opravili še test, ki je trajal 24h pri oknu 10.

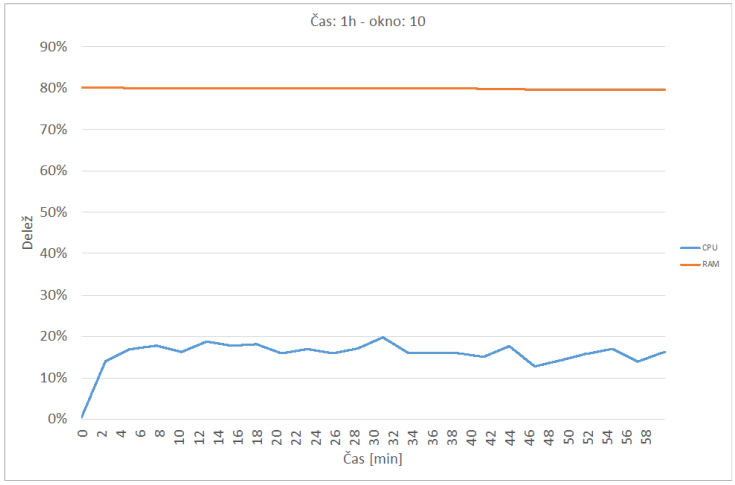
3.9.1 Testi 1h

Pri testu z oknom velikim 10 je zakasnitev konstantna. Strežnik vrne vse odgovore med 62 in 67 *ms* (glej sliko 3.16). V času ene ure smo poslali 548684 zahtevkov in strežnik je uspešno odgovoril na vse. V tem času je poraba pomnilnika stalno bila 800MB (80% od 1GB). Poraba procesorja se je postopoma dvignila iz 0% in nihala okoli 20% (glej sliko 3.17).

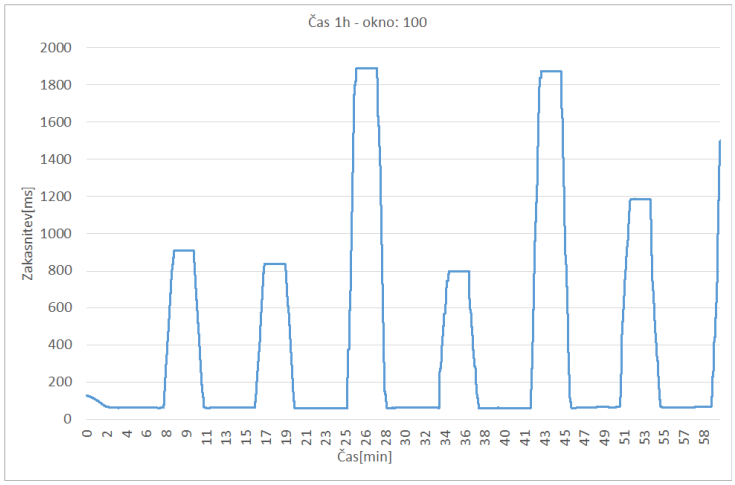
Slika 3.15: Graf zakasnitev 100 zahtev pri oknu velikem 10 z različno dolgimi datotekami. Lokacija: Bela Krajina; server: cloud.



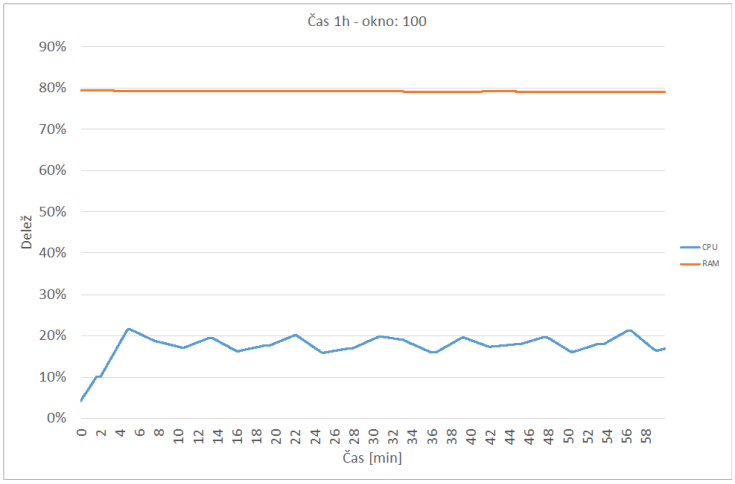
Slika 3.16: Graf poteka zakasnitev pri 1h obremenitve in oknu 10. Lokacija: Ljubljana; server: cloud.



Slika 3.17: Graf deleža porabe resursov pri 1h obremenitve in oknu 10. Lokacija: Ljubljana; server: cloud.



Slika 3.18: Graf poteka zakasnitev pri 1h obremenitve in oknu 100. Lokacija: Ljubljana; server: cloud.



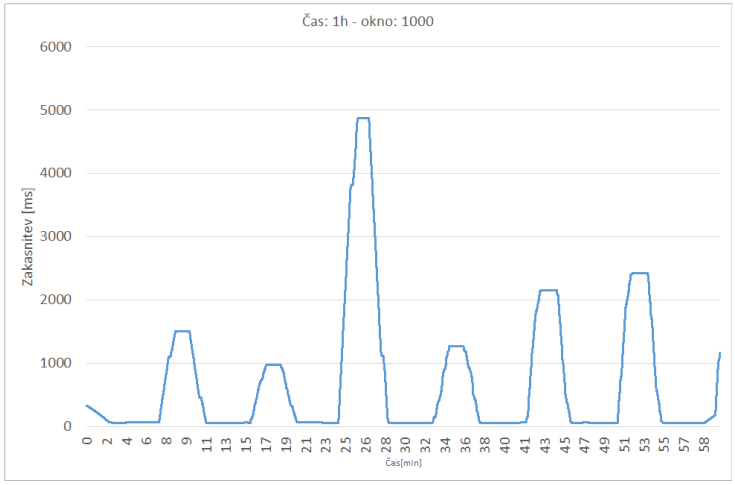
Slika 3.19: Graf deleža porabe resursov pri 1h obremenitve in oknu 100. Lokacija: Ljubljana; server: cloud.

Pri testu z oknom velikim 100 strežnik ni več uspel odgovoriti na vseh 616443 zahtevkov. Odjemalec ni prejel odgovora na 3080 (0,5%) zahtevkov. To lahko jasno razberemo iz slike 3.18, ko zakasnitve hitro zrastejo. Zahtevki, ki jih sistem uspe obdelati, imajo zakasnitev okoli 70ms. Poraba procesorja in pomnilnika se ni bistveno spremenila od prejšnjega testa pri oknu 10 (glej sliko 3.19).

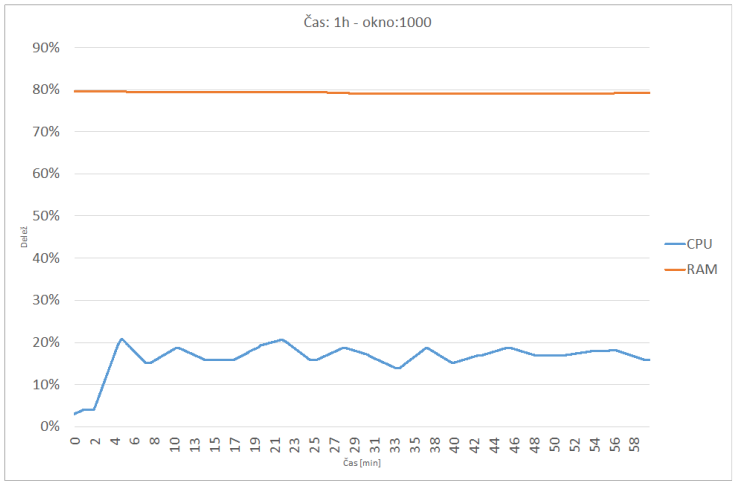
Zadnji enourni test smo opravili pri oknu 1000. Tokrat smo poslali 623304 zahtevkov, vendar strežnik ni uspel odgovoriti na 154523 (25%) izmed teh. Zahtevki brez odgovorov se na sliki kažejo kot povečane zakasnitve (slika 3.20). Potek porabe resursov je zopet ostal nespremenjen (glej sliko: 3.21). Iz vseh treh grafov porabe sklepamo, da programska oprema, ki teče na strežniku, neučinkovito uporablja sistemske resurse. Zaradi tega ne moremo predvideti zmogljivosti storitve na podlagi porabe resursov.

3.9.2 Testi 24h

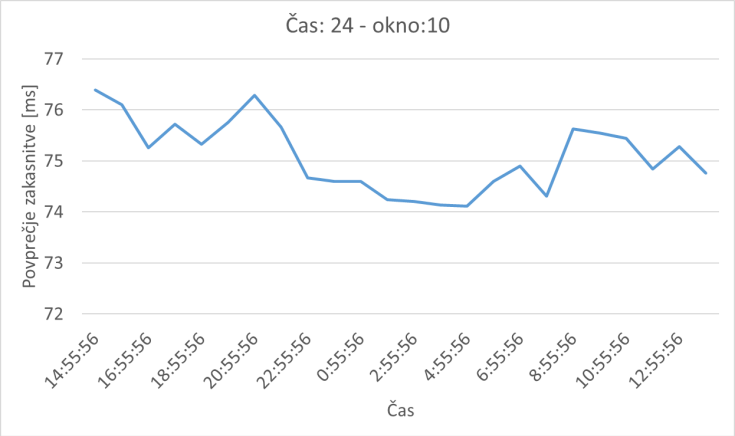
Opravili smo test, ki je trajal 24h pri oknu 10. Teste smo ponovili večkrat in izrisali grafe povprečnih zakasnitev. Tako smo lahko primerjali zakasnitve in določili razmere na omrežju ob določenem času. Merili smo iz Ljubljane in Bele Krajine. Iz slik 3.22 in 3.23 vidimo, da so največje zakasnitve na omrežju zvečer ob osmih, najnižje pa od enajstih do sedmih zjutraj. Sklepamo, da največ ljudi omrežje uporablja zvečer, najmanj pa ponoči.
Rezultate testov smo primerjali s podatki na Internet Traffic Report. To je spletna stran, ki meri pretok podatkov po svetu. Glede na njihove rezultate so v evropi največje zakasnitve ob 20:00, vendar njihove meritve ne kažejo nižjih zakasnitev ponoči [28].



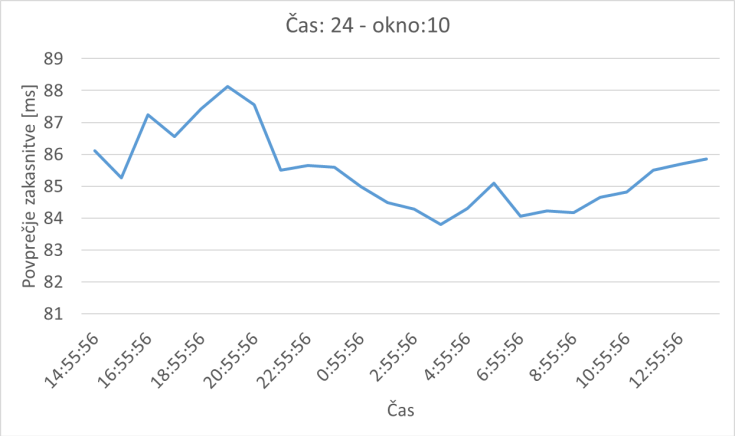
Slika 3.20: Graf poteka zakasnitev pri 1h obremenitve in oknu 1000. Lokacija: Ljubljana; server: cloud.



Slika 3.21: Graf deleža porabe resursov pri 1h obremenitve in oknu 1000. Lokacija: Ljubljana; server: cloud.



Slika 3.22: Graf deleža porabe resursov pri 24h obremenitve in oknu 10. Lokacija: Ljubljana; server: cloud.



Slika 3.23: Graf deleža porabe resursov pri 24h obremenitve in oknu 10. Lokacija: Bela Krajina; server: cloud.

3.10 Zaključek

V seminarski nalogi smo testirali zmogljivost delovanja spletne aplikacije Fear Index, napisane v izvajalnem okolju Node.js. Namestili smo jo na domačem strežniku v Škofji Loki in v oblaku pri ponudniku Dedimax. Izbrali smo ponudnikov najbolj osnovni paket. Teste smo opravljali iz dveh lokacij. Izkazalo se je, da na zakasnitve najbolj vplivajo razmere na omrežju in ne dolžina besedila, ki ga mora aplikacija pregledati. Testi iz Ljubljane so imeli občutno nižjo zakasnitev kot enaki testi iz Bele Krajine(glej sliki 3.3 in 3.7). Tu se je tudi izkazala največja prednost domačega strežnika. Ta se je nahajal mnogo bližje kot strežnik v oblaku, ki je posledično imel tudi višje omrežne zakasnitve (glej sliki 3.2 in 3.12). Ponudnik oblačnih strežnikov vps.net tudi trdi, da oddaljenost strežnika vpliva na zakasnitve. Te so nižje, če se strežnik nahaja bližje odjemalcu [29]. Na zakasnitve vpliva tudi čas v dnevu, saj je omrežje različno obremenjeno ob različnih urah. Najbolj je zasedeno ob 20:00 najmanj pa med 23:00 in 7:00. Največja omejitev storitve je število vzporednih odjemalcev, kar smo simulirali s strategijo okna. Oblačni strežnik smo obremenili za eno uro. Test smo ponovili trikrat, vsakič z večjim oknom. Pri oknu 10 je strežnik uspel odgvoriti na vse zahtevke. Pri oknu 100 ni uspel odgovoriti na 0,5% zahtevkov, pri oknu 1000 pa kar na 25%. Pri vseh treh testih je poraba resursov na strežniku ostala nespremenjena.
Glede na rezultate lahko zaključimo, da je najbolj osnovni paket pri oblačnem ponudniku zadosti zmogljiv za sprejemljiv nivo obdelave zahtev do 100 vzporednih porabnikov. Ko se število odjemalcv veča, strežnik ne uspe obdelati vse več zahtevkov. Takšen nivo zmogljivosti je zadosti visok za lokalno eksperimentalno rabo. Če bi storitev začelo uporabljati več uporabnikov, bi jo morali nadgraditi. Lahko bi jo namestili na bolj zmogljivem strežniku ali na več strežnikih in bi uporabnike ustrezno razdelili glede na zasedenost. Če storitev gostujemo v oblaku, lahko enostavno razširimo zmogljivost strežnika. Ponudnik storitve nam doda resurse na obstoječo virtualko, zato nam ni treba ponovno namestiti celotnega sistema na nov strežnik [30].

Poglavje 4

Analiza zmogljivosti jezikov za zaledni sistem

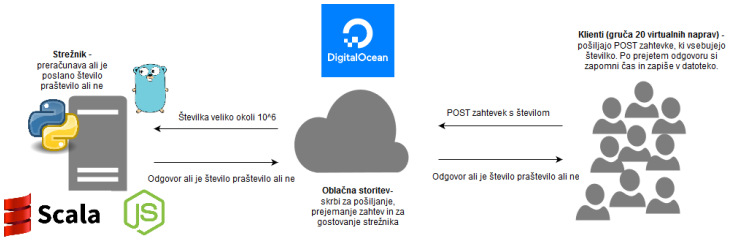
Juš Lozej, Jure Malovrh

4.1 Uvod

V današnjem času se razvoj spletnih storitev razvija s svetlobno hitrostjo. Storitve, ki so bile še pred letom nekaj novega, so postale zastarele, nadomeščajo pa jih nove. To lahko vidimo tudi pri razvoju spletnih strani. Včasih se je celoten izris spletne strani zgodil na strežniški strani. Strežnik nam je torej vrnil že pripravljeno in uporabniku unikatno spletno stran. Na ta način lahko ugotovimo, da je obremenitev strežnika, v času velikega obiska uporabnikov postala nevzdržna. V ta namen so se razvila različna orodja, ki nam omogočajo, da se celoten izris zgodi na uporabnikovi strani, strežniški del pa nam le še posreduje potrebne podatke, običajno skozi namenske API-je [31]. Prvotni zaledni jeziki so torej izgubili na svojem pomenu, zato so se razvili novi programski jeziki, ki imajo za glavno nalogo prav hitro izvajanje različnih API-jev. Primeri takih jezikov so Node.js [32], Go [33] itd.

4.2 Predstavitev ideje

V pričujočem razdelku opišemo izvedbo simulacije testiranja različnih programskih jezikov na zalednem sistemu. Glavna ideja je ta, da izvedemo teste hitrosti delovanja programov napisanih z različnimi programskimi jeziki, pri predpostavki, da vsi tečejo na primerljivih strojnih platformah. Potrebo po primerljivosti sistemov bo zagotovil oblačni sistem, saj predvidevamo, da je vsaka ustvarjena instanca popolnoma enaka drugim.



Slika 4.1: Shema testnega sistema.

Testiranje naše ideje bomo izvedli po naslednjih korakih: preko HTTP POST zahtevka bo odjemalec poslal sporočilo v formatu JSON. Sporočilo bo vsebovalo veliko naključno število. Oblačni sistem bo nato prejel sporočilo, iz njega prebral število in preveri ali je praštevilo. Rezultat svoje metode bo poslal v svojem odgovoru. Odjemalec bo prejel odgovor in si zapisal čas, ki ga je storitev potrebovala od časa poslanega paketa do prejema odgovora.

Glavna storitev naše aplikacije je torej testira ali je število, ki smo ga poslali praštevilo, ali ne. Ta izračun predstavlja neko breme, ki simulira realno okolje, kot je na primer dostop do baze ipd. Ne želimo si le vračati odgovorov oblike *Hello world*, kot je to v navadi pri mnogih testih.

Za namene testiranja smo pripravili štiri različne sisteme, napisane v različnih programskih jezikih: Node.js, Go, Python in Scala. Poleg zalednega sistema smo pripravili tudi odjemalca, ki bo pošiljal zahteve na strežnik. Pri testiranju smo merili različne metrike. Glavna je bila odzivni čas sistema (čas od poslanega paketa do prejetega odgovora). Ta nas zanima predvsem zato, ker nam manjši odzivni čas pomeni večjo prepustnost sistema, torej lahko naredi več odgovorov na sekundo. Merili smo tudi druge metrike, npr. porabo procesorja, RAM-a ipd.

Odjemalca smo implementirali v programskem jeziku Python. Njegova glavna naloga je pošiljanje zahtevkov na strežnik in merjenje časa, ki ga strežnik potrebuje za procesiranje ter odziv. Na ta način simuliramo odjemalca v realnem okolju, saj se večina podatkov v današnjem času prenese prav preko klicev na različne vmesnike API. Na sliki 4.1 je predstavljena shema testnega sistema.

4.3 Izbira ponudnika

Ponudnik, ki smo ga izbrali za našo strežniško stran aplikacije je DigitalOcean [34]. Za študente je začetna uporaba ponudnika brezplačna, saj nam nudi 50$ brezplačnega kredita. Poleg finančnih prednosti smo DigitalOcean izbrali predvsem zaradi dobre uporabniške podpore, ocen na spletu in enostavnosti uporabe (kdor je kadarkoli nastavljal Amazon AWS ve o čem je govora). Na našem ponudniku smo se odločili za drugo najcenejšo opcijo, katere značilnosti so sledeče:

- procesor: 1 jedro: Intel(R) Xeon(R) CPU E5-2650L v3 @ 1.80GHz,

- pomnilnik: 1 GB,
- disk: 30 GB.

Na ta način smo si zagotovili, da našim aplikacijam ne bo med izvajanjem prehitro zmanjkalo resursov.

Poleg strežniškega dela smo morali poskrbeti tudi za odjemalčev del. Za ta del je poskrbela gruča 20 virtualnih strežnikov, ki se nahajajo v Ljubljani. Gruča nam omogoča to, da hkrati poganjamo več instanc odjemalca in s tem bolj obremenimo strežnik. Hkrati imajo vse enake zmogljivosti in nam omogoča tudi boljše nadzorovanje frekvence poizvedb.

4.4 Izbira tehnologij

Za testiranje smo si izbrali več tehnologij. Glavni vodili razvoja sta bili enostavnost in možnost večkratne uporabe. Kot naše glavne strežniške jezike smo si izbrali Node.js, Scala, Golang in Python. Vsakega od navedenih jezikov bomo tudi natančneje predstavili. Tudi odjemalca smo spisali v enem od teh jezikov in sicer v Python-u.

4.4.1 Node.js

Node.js je odprtokoden JavaScript tolmač (interpreter). Temelji na Googlovem V8 JavaScript pogonu [35], ki deluje tako, da nam kodo napisano v programskem jeziku JavaScript prevede v strojno kodo. Glavna prednost Node.js-a pred ostalimi tekmeci je predvsem njegova dogodkovna arhitektura (angl. *event-driven architecture*), ki je zmožna asinhronih I/O operacij. Na ta način lahko optimiziramo prepustnost sistema in njegovo skalabilnost predvsem pri aplikacijah, ki imajo veliko I/O operacij ali pa se izvajajo v realnem času. Prednost uporabe Node.js-a je tudi v njegovem velikem razvojnem ekosistemu in je tako na voljo veliko število knjižnic, ki so dostopne preko platforme "npm"[36].

4.4.2 Scala

Scala [37] je splošno namenski programski jezik, ki podpira paradigmo funkcijskega programiranja in močno tipiziranost. Zgrajena je na platformi Java Virtual Machine (JVM) [38], kar pomeni, da je Scala interoperabilna z Java knjižnicami. To pomeni, da so nam v primeru uporabe jezika na voljo vse knjižnice, ki so že napisane v Javi. Ker je zgrajena v funkcijski paradigmi, so nam na voljo tudi vsi funkcijski "bonbončki", kot je npr. lena evalvacija, "pattern matching", ipd.

4.4.3 Go lang

Go [33] je najmlajši izmed izbranih programskih jezikov. Predstavljen je bil v letu 2009, implementirali pa so ga Googlovi inženirji. Jezik je bil razvit predvsem za namene strežniških sistemov, kjer je potrebna robustnost in hitrost.

Je jezik, čigar kodo prevajamo s statično tipiziranostjo tipov. Trenutna glavna slabost Go-ja je majhnost njegovega ekosistema, saj je v primerjavi z Node.js uporabnikov precej manj, s tem pa je manj tudi različnih knjižnic itd. Vendar pa v zadnjem času število uporabnikov strmo narašča, kar nam daje optimizem za nadaljnji razvoj ekosistema jezika.

4.4.4 Python

Python [39] je najstarejši izmed uporabljenih jezikov. Je splošno namenski, visokonivojski jezik. Ker je interpretiran jezik, nima težav z okolji in prevajalniki. Glavna filozofija, na kateri temelji, je berljivost kode. To filozofijo implementira v obliki standarda PEP [40], ki vsebuje pravila kako oblikovati kodo, kakšne oblike morajo biti funkcije ipd. Ker je jezik med nami že precej časa, ima veliko uporabnikov in s tem tudi veliko število kakovostnih knjižnic.

4.5 Implementacija sistema

Pričujoče poglavje se osredotoča na implementacijo našega sistema. V podpoglavjih natančneje opišemo razdelitev aplikacije na strežniško in odjemalčevo infrastrukturo ter opišemo, kako smo nadzorovali sistem.

4.5.1 Strežniška aplikacija

Pripravili smo strežniško aplikacijo napisano v štirih različnih jazikih, ki jih uporabimo za zaledni sistem. Specifikacije vseh aplikacij so bile enake. Vse aplikacije so delovale po arhitekturnem načelu REST storitev [41]. Aplikacije navzven ponujajo POST API, ki je nato klican s strani odjemalca.

Odjemalec ob klicu storitve v telo metode doda številko. Za uporabo številke obstaja več razlogov. Glavni razlog je ta, da na ta način prisilimo strežnik v vsakokratni izračun. To pomeni, da si strežnik ne more hraniti odgovorov, saj ne bo vedel kakšen odgovor se od njega pričakuje. Na ta način lahko precej bolj realno ocenimo zmogljivost sistema. S tem pa lahko bolje ocenimo tudi hitrost izvajanja jezika, saj si v primeru velike obremenitve naše strani želimo hitrejše izvajanje kode programskega jezika.

Pri našem testu smo si želeli izogniti različnim ozkim grlom, ki se v razvoju današnjih spletnih aplikacij pojavijo. V večini primerov je pisanje iz in v bazo v primerjavi z ostalim delovanjem aplikacije največja ovira. Za primerjavo, če pretvorimo 1 CPU cikel, ki ga navadno merimo v *ns* na velikost 1 sekunde, potem je dostop do baze, ki je običajno na hitrih SSD diskih, merjen v dnevih (od 2 do 6 dni) [42]. K počasnosti vseh teh storitev prinese tudi implementacija različnih gonilnikov, saj nimamo zagotovila optimalnosti. Izognili smo se tudi pisanju in branju iz različnih datotek. Nismo pa želeli implementirati le osnovne t.i. "Hello world"aplikacije, saj se nam zdi, da nam tak način aplikacije ne ponudi dovolj pokazateljev zmogljivosti sistema.

4.5.2 Odjemalec

Odjemalec je napisan v jeziku Python. Glavna naloga aplikacije je, da pošilja POST zahtevke na strežnik in meri čas, ki je minil med poslanim zahtevkom in prejetjem odgovora. Med samim delovanjem si shranjuje čas posamezne meritve, na koncu pa se izračuna še povprečni čas, maksimalni čas in minimalni čas. Vse meritve se tudi zapisujejo v datoteko CSV. Ker se naši testi izvajajo paralelno, je bilo za izris podatkov in njihovo analizo podatke potrebno združiti. To smo naredili tako, da smo za čas vzeli povprečne vrednosti vseh izmerjenih podatkov, ki so se zgodile ob istem času, minimum in maksimum pa sta postala najmanjša in največja vrednost vseh meritev.

4.6 Breme storitve

Breme storitve je v našem primeru nek enostaven izračun. V zahtevku sitem prejme število, ki ga vzamemo za breme, to je število velikosti okoli 10^6. Za število nato preverimo, ali je praštevilo ali ni. Po končanem preverjanju sistem vrne odgovor ali je podano število praštevilo. Zato, da bi odjemalec lahko preveril ali je preverba pravilna ali ni, si bomo nekaj večjih števil izračunali vnaprej in jih shranili v slovarje, ki omogočajo hiter dostop.

4.7 Nadzor sistema

Ker smo poleg hitrosti odgovorov merili tudi obremenitev fizičnega sistema kot takega, smo preverjali tudi vrednosti različnih metrik delovanja sistema. Za pregledovanje sistema smo uporabili sistem, ki je v DigitalOcean že vključen. Vmesnik prikazuje osnovne podatke o sistemu, kot so poraba pomnilnika, CPU-ja, I/O operacij ipd. Videz sistema za pregledovanje lahko vidimo na sliki 4.2.

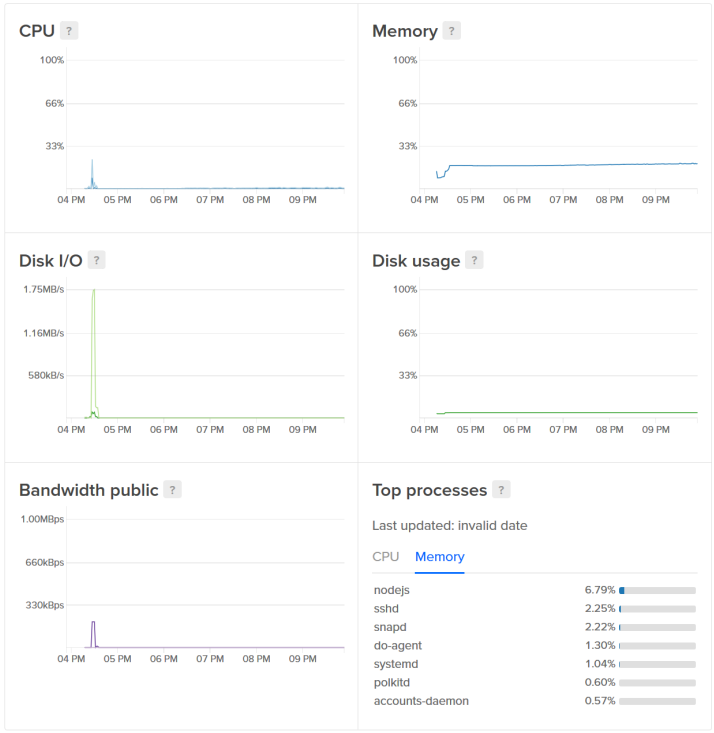
4.8 Izbrane metrike

Pri testiranju smo merili naslednje metrike:

- odzivni čas,
- zasedenost računalniških resursov (CPU, RAM),
- maksimalno obremenitev sistema.

4.8.1 Odzivni čas

Odzivni čas je čas, ki je potekel od pošiljanja zahtevka do prejema odgovora. Pove nam okvirno zmogljivost strežnika in merimo ga v milisekundah. Z povprečnim odzivnim časom lahko izračunamo tudi prepustnost sistema. Prepustnost nam pove koliko odgovorov lahko pričakujemo v časovnem segmentu. Prepustnost merimo v odgovorih na sekundo.



Slika 4.2: Grafični vmesnik za nadzor.

4.8.2 Stanje računalniških resursov

Med testiranjem smo bili pozorni tudi na resurse sistema, torej zasedenost procesorja, porabo pomnilnika. Pri tem smo spremljali kakšna je poraba ob mirovanju sistema in tako merili razliko v obremenitvi. Namen testa je, da najdemo jezik, katerega koda med izvajanjem porabi najmanj sistemskih resursov in je tako primeren za manj zmogljive sisteme.

4.8.3 Maksimalna obremenitev sistem

Maksimalna obremenitev sistema nam pove koliko zahtev moramo poslati, da sistem odpove oz. postane odzivni čas pretirano dolg. Zanima nas ali bo jezik, katerega koda ima med izvajanjem najmanjši odzivni čas premogel tudi največ zahtev. Izpade smo merili na dva načina z velikim sunkom zahtev ter z visoko konstantno frekvenco pošiljanja.

4.9 Testiranje

Testiranje smo razdelili na 2 dela, sinhroni in asinhroni del. V vsakem delu smo testiranje nadaljnjo razdelili še na 3 dele.

V prvem, sinhronem delu smo testiranje izvedli tako, da smo uporabili različno število odjemalcev. Strategija pošiljanja zahtev je bila v prvem delu izvedena tako, da je vsak odjemalec poslal en zahtevek, počakal na odgovor strežnika, izračunal odzivni čas, ter ga shranil v datoteko. število uporabljenih klientov se razlikuje glede na posamezen del.
V prvem podrazdelku smo naš sistem testirali s 5 odjemalci, ki so sočasno klicali API. V drugem podrazdelku smo testirali z 10 odjemalci in v tretjem z 20 odjemalci. Pričakovali smo predvsem večji odzivni čas pri testiranjih z več odjemalci. Po končanem testiranju smo rezultate, ki smo jih pridobili v datoteki CSV združili (istoležne vrstice smo povprečili), izračunali smo povprečni odzivni čas, ter minimalni in maksimalni čas, ki smo ga potrebovali za odgovor. Več o samih rezultatih lahko preberemo v razdelku 4.10.

V drugem glavnem delu smo testiranje izvedli tako, da smo iz 20 klientov naenkrat pošiljali več kot 1 zahtevek. Strategija pošiljanja v tem primeru je bila izvedena tako, da smo s pomočjo knjižnice sprožili več asinhronih klicev. Po končanem pošiljanju smo izmerili čas, ki ga je strežnik potreboval za odgovor na vse klice. V tem primeru smo strežnik obremenili precej bolj učinkovito. V prvem podrazdelku smo naš sistem testirali s 100 asinhronimi klici, torej smo skupaj poslali 2000 zahtevkov. V drugem podrazdelku smo sistem testirali s 500 asinhronimi klici, torej smo poslali 10000 zahtevkov. V tretjem podrazdelku pa smo sistem testirali s 1000 asinhronimi klici, torej smo poslali 20000 zahtevkov. Rezultate smo zbrali v datoteki CSV, kjer smo rezultate prav tako povprečili, kot smo to storili v prvem glavnem delu. Prav tako smo izračunali povprečni čas, minimalni in maksimalni čas. Dodatno smo izračunali tudi uspešnost klicev, torej delež klicev, ki se končajo s pozitivnim odgovorom.

Poleg obeh glavnih testiranj, smo dodali tudi 24 urno testiranje. Pri tem testiranju smo strežnike s konstantno obremenitvijo obremenjevali 24 ur zapored. S tem smo želeli preveriti, ali ima koda, katerega od programskih jezikov skrito kakšno skrito napako, ki povzroča prekomerno porabo pomnilnika (angl. *memory leak*). Poleg pomnilnika smo opazovali tudi gibanje CPU, saj nas je zanimalo, ali bo poraba konstantna ali pa se bo skozi dan spreminjala. Slike in komentar se nahajajo v razdelku 4.11.3.

Testiranje smo izvajali med tednom (od ponedeljka do četrtka), ob različnih časovnih intervalih. Testiranje smo izvajali le med ponedeljkom in četrtkom zato, da smo dobili najbolj realne rezultate. Izogniti smo se želeli višjim obremenitvam, ki jih prinese konec tedna, saj je več ljudi prostih in je zato tudi zasedenost omrežja višja. Glavni čas ob katerem smo testirali je bil dopoldanski (med 11:00 in 14:00) in večerni čas (med 20:00 in 23:00). Vsako testiranje je trajalo okoli 10 minut (približno 5000 klicev).

4.10 Rezultati meritev

V tem razdelku predstavimo rezultate izvajanja posameznega jezika za zaledni sistem. V vsakem podrazdelku predstavimo graf izvajalne kode različnih jezikov, ki kaže odzivni čas. Poleg tega v tabeli predstavimo tudi povprečen, minimalen in maksimalen čas, ki ga je zaledni čas potreboval za posredovanje odgovora. Maksimalen čas je v našem primeru povprečje 5 najvišjih maksimalnih časov. Na ta način smo si želeli zagotoviti, da je maksimalen čas bolj realen, ne le posledica enega posameznega skoka. V tabeli je dodan tudi izračun prepustnosti sistema, ki je izračunana po izrazu

$$\frac{1000}{avg\_time(responses)}.$$

4.10.1 Sinhrono testiranje

Node.js

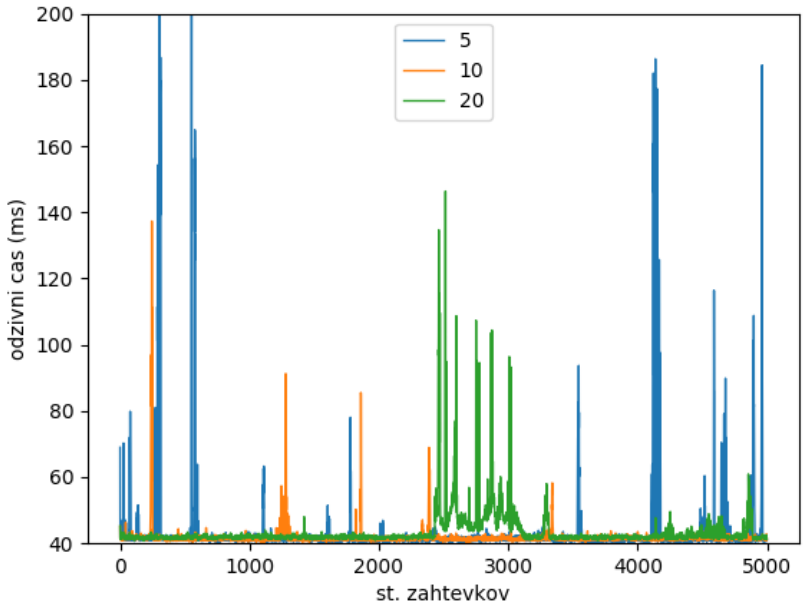
Numerični rezultati za jezik Node.js so prikazani v tabeli 4.1. Graf odzivnih časov pri 5, 10 in 20 odjemalcih je viden na sliki 4.3. Kot je opazno so odzivni časi precej enakomerni, pri več odjemalcih pa se lahko zgodi da se čas precej podaljša, kar vidimo tudi na grafih. Poleg tega lahko vidimo v primeru 5 odjemalcev tudi nihanje omrežja, saj so odzivni časi tam najmanj enakomerni.

Python

Numerični rezultati za jezik Python so prikazani v tabeli 4.2. Grafi odzivnih časov pri 5, 10 in 20 odjemalcih so vidni na sliki 4.4. Pri več odjemalcih lahko opazimo, da se čas posameznih izbruhov precej podaljša (predvsem pri številu odjemalca 10 in 20). Poleg tega vidimo, da se čas v primeru več odjemalcev podaljša takoj ob začetku testiranja in se kasneje umiri.

število odjemalcev	povprečni o. č. (ms)	minimalni o. č. (ms)	maksimalni o. č. (ms)	prepustnost sistema (zahtevko-v/s)
5	43.39	39.34	277.83	23.10
10	41.76	39.26	179.06	23.90
20	43.14	39.24	1055.24	23.21

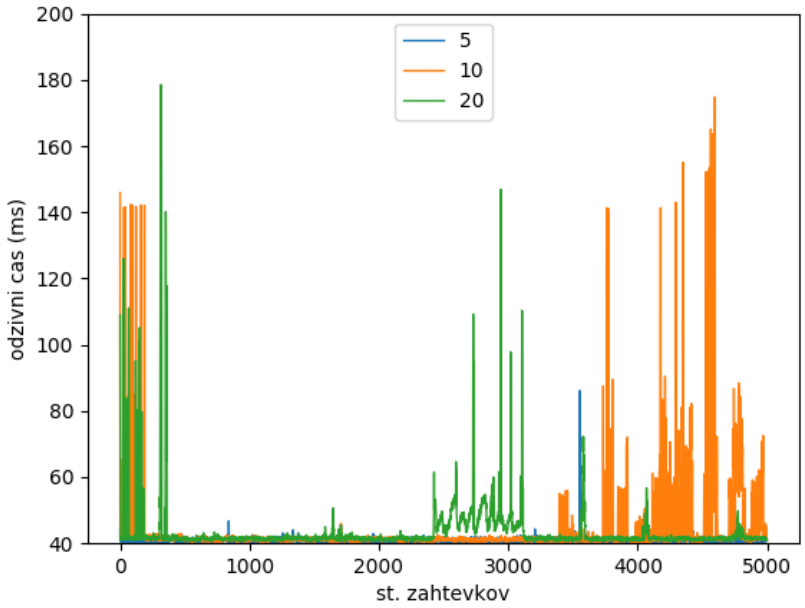
Tabela 4.1: Numerični rezultati programskega jezika Node.js (o. č. = odzivni čas).



Slika 4.3: Graf odzivnih časov jezika Node.js, pri sinhronem testiranju.

število odjemalcev	povprečni o. č. (ms)	minimalni o. č. (ms)	maksimalni o. č. (ms)	prepustnost sistema (zahtevko-v/s)
5	40.96	38.96	94.73	24.40
10	43.53	38.97	1157.79	23.00
20	43.15	38.94	1046.60	23.10

Tabela 4.2: Numerični rezultati programskega jezika Python (o. č. = odzivni čas).



Slika 4.4: Graf odzivnih časov jezika Python pri sinhronem testiranju.

število odjemalcev	povprečni o. č. (ms)	minimalni o. č. (ms)	maksimalni o. č. (ms)	prepustnost sistema (zahtevkov/s)
5	42.06	39.50	145.08	23.80
10	41.67	39.29	64.34	24.00
20	43.10	39.13	1044.60	23.20

Tabela 4.3: Numerični rezultati programskega jezika Scala (o. č. = odzivni čas).

število odjemalcev	povprečni o. č. (ms)	minimalni o. č. (ms)	maksimalni o. č. (ms)	prepustnost sistema (zahtevkov/s)
5	41.39	38.95	249.85	24.10
10	40.85	38.89	58.67	24.50
20	42.94	38.94	1160.10	23.30

Tabela 4.4: Numerični rezultati programskega jezika Go (o. č. = odzivni čas).

Scala

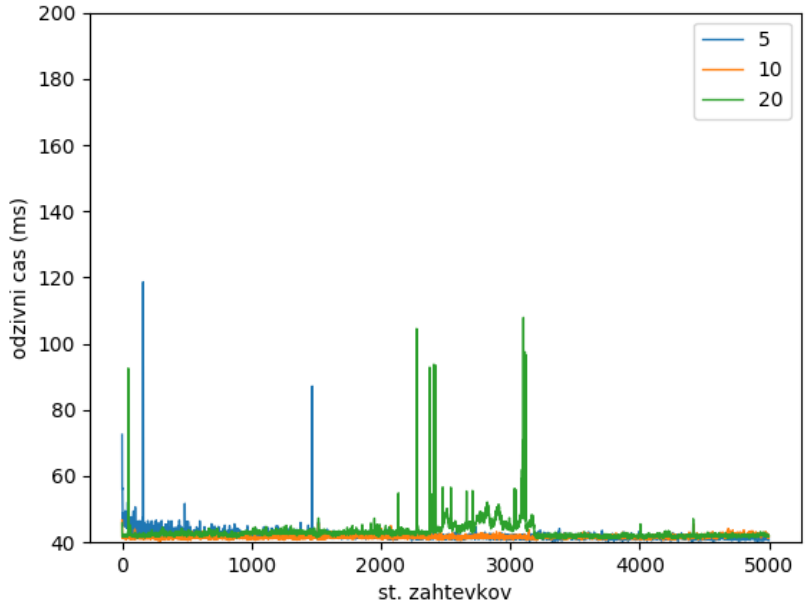
Numerični rezultati za jezik Scala so prikazani v tabeli 4.3. Grafi odzivnih časov pri 5, 10 in 20 odjemalcih so vidni na sliki 4.5. Kot je lahko opazno na grafih, je odzivni čas precej enakomeren in ni veliko izbruhov, kjer bi se odzivni čas precej podaljšal.

Go

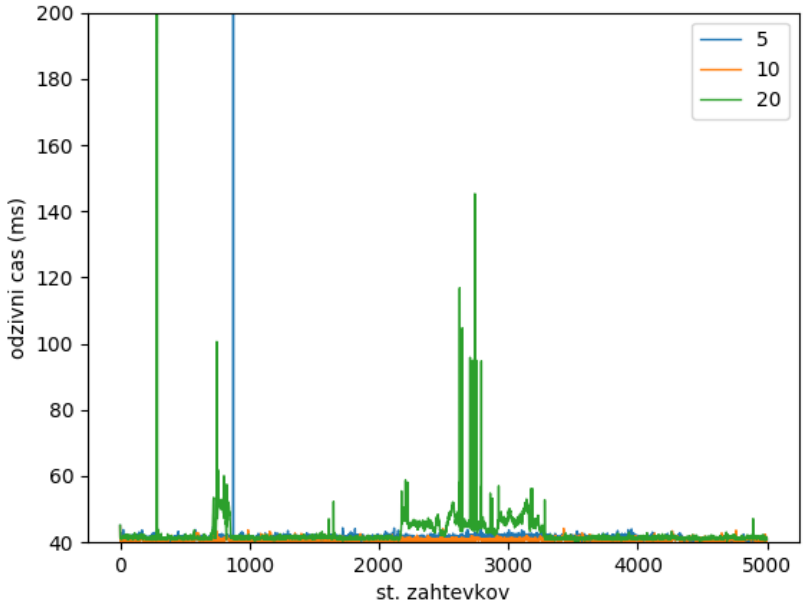
Numerični rezultati za jezik Go so prikazani v tabeli 4.4. Grafi odzivnih časov pri 5, 10 in 20 odjemalcih so vidni na sliki 4.6. Na prvi pogled so grafi pri pri programskem jeziku Go najbolj enakomerni. V primeru 20 odjemalcev se tudi tukaj zgodi manjši izbruh.

4.10.2 Asinhrono testiranje

Pri asinhronem testiranju, smo merili čas, ki ga potrebuje klient, da opravi vse klice. Povprečni čas za odgovore smo prikazali v enem grafu, saj lahko tako najlažje primerjamo uspešnost. Pri vsakem jeziku smo dodali tudi numerične izračune. Poleg izračunov in grafov, smo dodali tudi uspešnost posameznega jezika t.j. kolikšen odstotek paketov je prišel nazaj na svoj cilj. Seveda si želimo, da bi naši programi napisani z različnimi jeziki imeli čim višjo uspešnost.



Slika 4.5: Graf odzivnih časov jezika Scala pri sinhronem testiranju.



Slika 4.6: Graf odzivnih časov jezika Go pri sinhronem testiranju.

število klicev	povprečni o. č. (ms)	minimalni o. č. (ms)	maksimalni o. č. (ms)	uspešnost prejema (%)
100	965.42	211.44	12128.27	100.00
500	5757.47	1033.34	64333.96	98.10
1000	12457.26	2141.43	141336.91	97.10

Tabela 4.5: Numerični rezultati programskega jezika Node.js pri asinhronem testiranju (o. č. = odzivni čas).

število klicev	povprečni o. č. (ms)	minimalni o. č. (ms)	maksimalni o. č. (ms)	uspešnost prejema (%)
100	876.32	183.41	14121.83	98.30
500	4978.87	1073.34	62196.19	97.60
1000	10862.63	2462.13	169532.59	97.20

Tabela 4.6: Numerični rezultati programskega jezika Python pri asinhronem testiranju (o. č. = odzivni čas).

Node.js

Numerični rezultati pri asinhronem testiranju so prikazani v tabeli 4.5. Graf odzivnih časov je viden na sliki 4.7. Kot lahko vidimo na prvi pogled, se vrednost odzivnega časa dviguje precej linearno. Tudi uspešnost je pri manjšem številu klicev kar 100%. Tudi pri višji obremenitvi ostaja program precej robusten.

Python

Numerični rezultati pri asinhronem testiranju so prikazani v tabeli 4.6. Graf odzivnih časov je viden na sliki 4.7. Kot vidimo je čas precej podoben časom pri jeziku Node.js, je pa robustnost sistema malce manjša, saj ima že pri najmanjšemu številu klicev določeno izgubo, torej neprejete odzive na strani odjemalca.

Scala

Numerični rezultati pri asinhronem testiranju so prikazani v tabeli 4.7. Graf odzivnih časov je viden na sliki 4.7. Opazimo lahko, da robustnost ni popolna, saj ima že pri najmanjšem številu odjemalcev odjemalec občuti določene izgubljene odzive.

Go

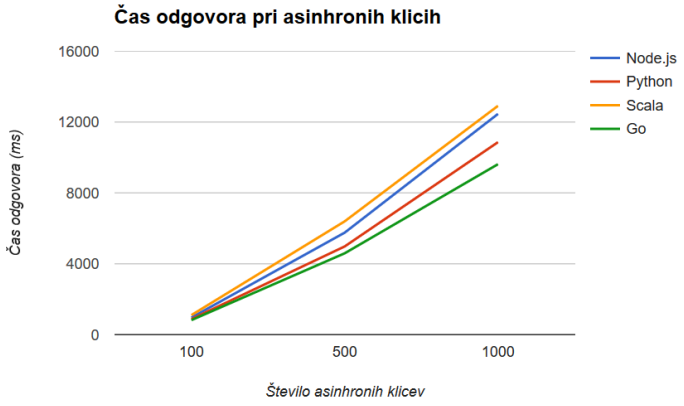
Numerični rezultati pri asinhronem testiranju so prikazani v tabeli 4.8. Graf odzivnih časov je viden na sliki 4.7. Kot pričakovano, so rezultati med boljšimi. Tudi zanesljivost odzivanja je skoraj 100%.

število klicev	povprečni o. č. (ms)	minimalni o. č. (ms)	maksimalni o. č. (ms)	uspešnost prejema (%)
100	1103.12	316.44	14131.27	97.10
500	6395.54	1015.32	67105.12	96.30
1000	12912.98	2512.24	172877.04	95.70

Tabela 4.7: Numerični rezultati programskega jezika Scala pri asinhronem testiranju (o. č. = odzivni čas).

število klicev	povprečni o. č. (ms)	minimalni o. č. (ms)	maksimalni o. č. (ms)	uspešnost prejema (%)
100	821.18	186.44	11516.15	100.00
500	4591.94	976.72	52312.98	99.60
1000	9620.10	1947.43	185225.75	98.80

Tabela 4.8: Numerični rezultati programskega jezika Go pri asinhronem testiranju (o. č. = odzivni čas).



Slika 4.7: Graf odzivnih časov pri asinhronem testiranju.

4.11 Komentarji rezultatov

4.11.1 Sinhrono testiranje

Kot lahko vidimo na rezultatih, so ti med seboj povsem primerljivi. Iz danih rezultatov v primeru sinhronega testiranja ne moremo določiti, katera programska koda se izvaja najhitreje oz. najbolj učinkovito. To je verjetno predvsem posledica tega, da je obremenitev pri sinhronem testiranju premajhna, da bi dovolj obremenila posamezen sistem. Kar je zanimivo je tudi to, da je v vseh štirih primerih prišlo vsaj do enega izbruha, kjer so se odzivni časi povečali.

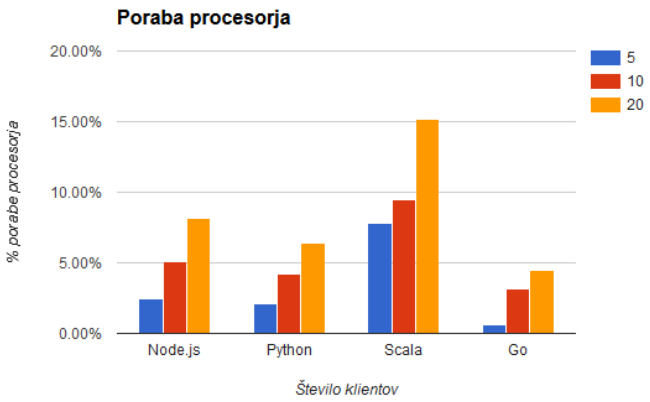
Glede na grafe, se je kot najbolj konsistentna izkazala koda, ki je bila napisana v programskem jeziku Scala. Ima najmanjša nihanja in tudi izbruhi so precej mili (v primeru 10 odjemalcev jih sploh ni). Druga sistem, ki se je izkazala kot precej konsistenten je sistem napisan v jeziku Go. Tudi v tem primeru so nihanja precej nizka, vendar pa je v primeru 20 odjemalcev opazen del, kjer se odzivni čas poveča. Ti rezultati niso presenetljivi, saj je programska koda, napisana v teh jezikih, prevedena v zbirno kodo, ki je optimizirana procesorski arhitekturi računalnika, ki poganja sistem.

Poraba sistemskih resursov je največja prav pri izvajalni kodi jezika Scala. Tudi to nas ne preseneča, saj temelji na virtualnem stroju JVM, ki ni ravno varčen z računalniškimi viri. Predvsem se to opazi pri porabi spomina RAM, kar lahko vidimo na sliki 4.9. V tem primeru je poraba spomina RAM dvakrat večja od preostalih testiranih jezikov. Najmanj spomina je za svoje delovanje porabil Python, kar je presenetljivo, saj je jezik tolmačen.
Tudi v primeru porabe procesorja se je Scala odrezala najslabše. Rezultate porabe procesorja lahko vidimo na sliki 4.8. V primeru porabe procesorja je procesor najmanj obremenila izvajalna koda jezika Go. To ni presenetljivo, saj vemo, da je bil Go napisan z namenom minimizirati porabe računalniških resursov velikih obremenitev.

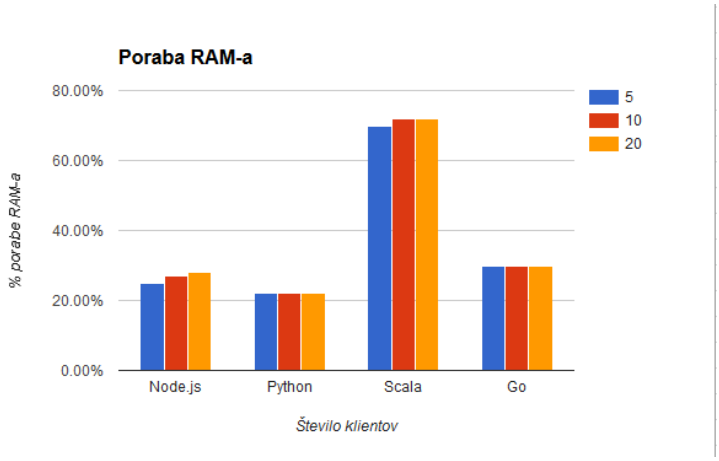
4.11.2 Asinhrono testiranje

Kot je razvidno iz rezultatov, se pri večji obremenitvi sistema začnejo pojavljati večje razlike med jeziki, kar je vidno v njihovem odzivnem času in tudi v porabi resursov. Iz asinhronih testov lahko opazimo, da ima najmanjši povprečni odzivni čas programski jezik Go. Na grafu lahko tudi opazimo, da odzivni čas jezikov pri vseh primerih narašča precej linearno, torej je v primeru z 1000 klienti približno 10x večji, kot je v primeru s 100 klienti. Opazimo lahko tudi, da pri jeziku Scala, ki je bila pri sinhronem testiranju najbolj konsistentna, ne zdrži velikega pritiska. Njen odzivni čas je najpočasnejši med vsemi.

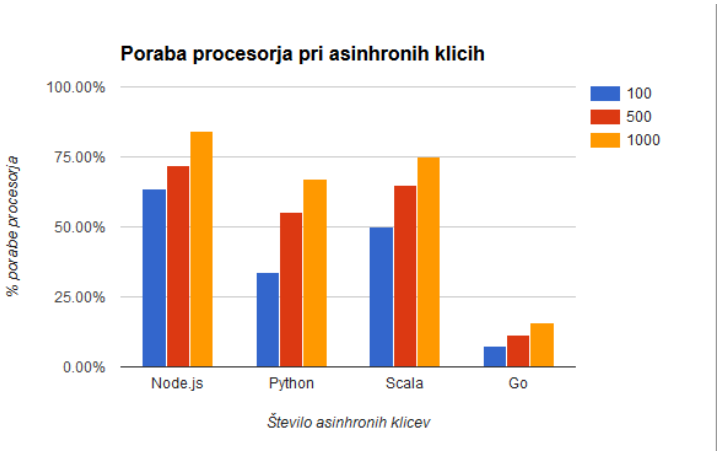
Poraba sistemskih resursov je precej zanimiva, saj se med jeziki opazi precejšnja razlika. Kot je pričakovano, je med jeziki najbolj uspešen jezik Go. To nas ni presenetilo, saj smo že pri sinhronem testiranju omenili, da je bil napisan z namenom minimizirati porabo sistemskih virov. V porabi procesorja je zelo očitna razlika, saj njegova koda med izvajanjem porabi od 5 do 10 krat manj procesorske moči, kar je razvidno v grafu 4.10. Tudi v porabi spomina RAM je



Slika 4.8: Graf porabe procesorja med testiranjem pri različnem številu klientov.



Slika 4.9: Graf porabe RAM-a med testiranjem pri različnem številu klientov.



Slika 4.10: Graf porabe procesorja med asinhronim testiranjem.

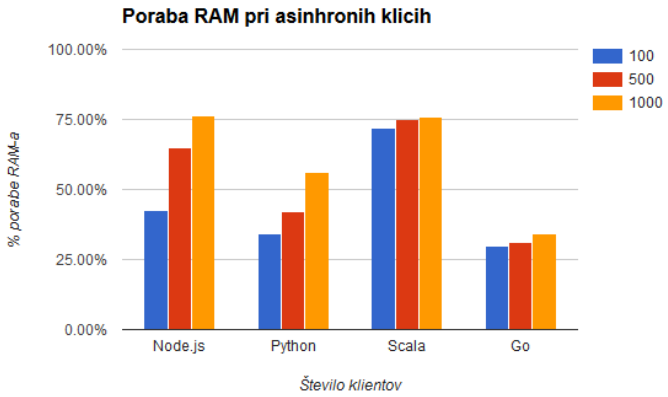
razlika očitna. Ostali rezultati so prikazani na 4.11.

Pomembna metrika, ki jo je potrebno omeniti je tudi robustnost jezika. Torej delež ne-procesiranih zahtevkov relativno na obseg prometa. Nočemo namreč, da bi se paketi zaradi daljšega časa obravnave izgubljali. če lahko sprejmemo občasen daljši odzivni čas v primeru velikega št. zahtev, pa ne želimo paketov izgubljati in tako povzročati ponovno pošiljanje ali izgubo podatkov ipd. Zato smo merili tudi robustnost izvajalne kode posameznega jezika. Tudi v tem primeru, se je kot najboljši izkazal programski jezik Go, vendar pa po uspešnosti tesno sledi programski jezik Node.js.
V primeru, da se odločamo med uporabo Node.js in Python je odločitev precej težavna, saj je Python malenkost hitrejši, Node.js pa malenkost bolj uspešen pri procesiranju zahtev.

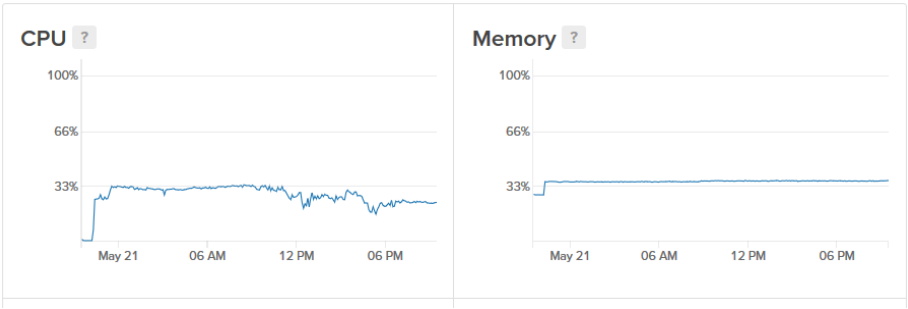
4.11.3 24 urno testiranje

Pri 24 urnem testiranju smo preverjali, ali ima kateri od jezikov skrite pomanjkljivosti, ki povzročijo t.i. memory leak-e, konstanto izgubljanje razpoložljivega spomina. Najbolje se ponovno obnese programski jezik Go (slika 4.15). Najslabše pa ponovno jezik Scala (slika 4.14). Preostala jezika Node.js (slika 4.12) in Python (slika 4.13) po rezultatih ne izstopata v nobeno smer. Na slikah nam y-os predstavlja odstotek porabe resursa, x-os pa nam predstavlja čas v katerem se je meritev zgodila.

Kot razvidno iz slik, je edini jezik, kjer pride do opazne spremembe razpoložljivega pomnilnika, Scala. Vendar ta sprememba ni rezultat memory leak-a. Sklepamo torej, da Scala pod dolgotrajno obremenitvijo, poskuša optimizirati pomnilnik, zato tudi vidimo spust, ki se je zgodil po približno treh urah testiranja. Spust zasedenega pomnilnika je lahko tudi rezultat zbiralca smeti (garbage collector). Na podlagi danih rezultatov, lahko zaključimo, da noben izmed te-



Slika 4.11: Graf porabe RAM-a med asinhronim testiranjem.

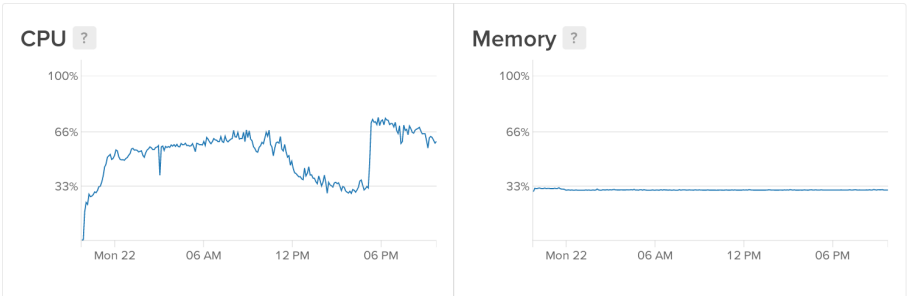


Slika 4.12: 24 urno testiranje jezika Node.js.

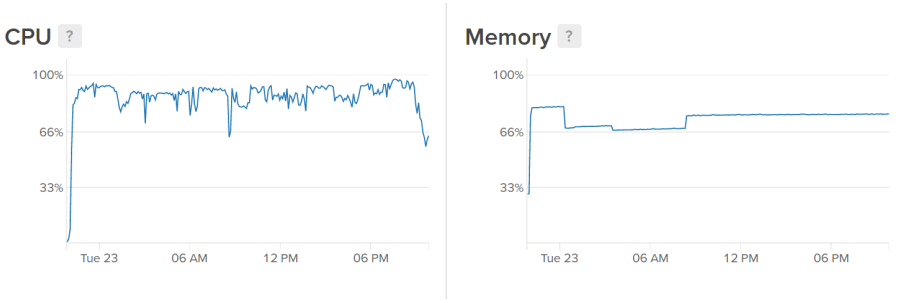
stiranih jezikov nima težav s pomnilnikom oz. slabim upravljanjem le tega.

Drugačna zgodba pa je v primeru procesorja. V tem primeru se ponovno najbolje obnese programski jezik Go, ki pa mu presenetljivo sledi programski jezik Node.js in ne Python, kot se je to pokazalo pri naših krajših enkratnih testiranjih. To je lahko rezultat tega, da se je testiranje programa implementiranega v jeziku Node.js izvajalo v nedeljo, preostali jeziki so pa bili testirani skozi teden, vendar pa bi bili potem tudi rezultati programskega jezika Go precej drugačni. Vidimo lahko tudi, da poraba procesorja v nobenem primeru ni enakomerna. To lahko pripišemo zasedenosti omrežja, saj vidimo, da mora strežnik delati manj oz. ima manjšo obremenitev ravno v popoldanskem času, ko je zasedenost omrežja najvišja.

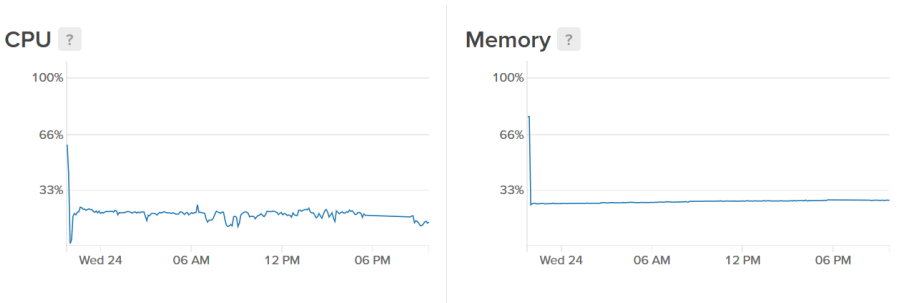
Zaključek 24 urnega testiranja je torej sledeče: pri nobenem od jezikov nam ni uspelo odkriti t.i. memory leak-a, obremenitev procesorja pa je odvisna od časa v dnevu, torej je večja, ko je omrežje manj zasedeno (v primeru, da imamo vedno konstantno obremenitev) in manjša, ko je omrežje bolj zasedeno.



Slika 4.13: 24 urno testiranje jezika Python.



Slika 4.14: 24 urno testiranje jezika Scala.



Slika 4.15: 24 urno testiranje jezika Go.

4.12 Zaključek

Namen raziskave je bilo v kontroliranem okolju testirati zanesljivost in učinkovitost izvajalnih kod posameznih programskih jezikov namenjenih pisanju strežniških arhitektur. Med izbranimi jeziki so bili tako starejši že velikokrat uporabljeni jeziki kot tudi popolnoma novi. Testi so bili izvedeni na jezikih Scala, Python, Node.js, ki je trenutni najbolj priljubljen jezik za te namene, ter Go, ki je najnovejši med naštetimi.

Pri testih se je najbolje odrezal prav jezik Go, saj je pokazal najboljše razmerje med robustnostjo in porabo sistemskih virov. Presenetilo nas je, da kljub eksponentnemu naraščanju zahtev je programova poraba spomina in procesorja le rahlo naraščala. Ko so programi ostalih programskih jezikov posegali že po 75 % spomina je Go potreboval le okoli 17 %. Pri dolgotrajnem testu tudi ni kazal nobenih dodatnih zahtev po resursih. Pri velikem št. zahtev jih je izgubil le zelo majhen delež (pri največji obremenitvi le 1.2%). Imel je tudi najboljše odzivne čase. Edina kritika jezika bi lahko bila, da zaradi novosti nima veliko že izdelanih knjižnic, vendar se to lahko v prihodnosti spremeni, saj je Go jezik, na katerega največ razvijalcev trenutno migrira.
Drugo mesto si delita jezik Node.js ter Python. Node je imel malce manj izgub zahtev, Python pa malenkost boljše odzivne čase ter porabo računalniških virov. Zaradi velike razvijalne skupnosti ter nabora izdelanih knjižnic se nam ob zdita jezika še vedno relevantna in uporabna.
Na testih se je najslabše odrezal jezik Scala. Uporabo jezika na podlagi izvedenih testov ne moramo priporočati.

Poglavje 5

Analiza zmogljivosti oblačne storitve Dropbox

Gašper Primožič, Sebastien Strban

5.1 Opis problema

Ker postajajo oblačne storitve vse bolj ekonomične in nekaj popolnoma običajnega, smo se odločili testirati zmogljivost ene izmed takih storitev. Ker živimo v času kjer nam vladajo informacije, si želimo, da so te informacije agregirane, varovane, nam pa vedno dostopne ko jih potrebujemo.

Iz teh, ali pa podobnih namenov, se velikokrat odločimo uporabiti Dropbox, zlasti zaradi razširjenosti, priljubljenosti in enostavnosti, sami pa se ne vprašamo in ne pomislimo, kam točno naši podatki gredo. Namen pričujočega poglavja je analizirati zmogljivost takega oblačnega ponudnika, na podlagi izmerjenih metrik. Npr.: hitrost dostopa, hitrost prenosa podatkov na oblak, hitrost prenosa podatkov iz oblaka,... S tem želimo ugotoviti zmožnosti oblačnega ponudnika. Na podlagi dobljenih meritev želimo ugotoviti prednosti ali pa pomanjkljivosti takih sistemov, v kolikor bi se odločili za uporabo takšne storitve.

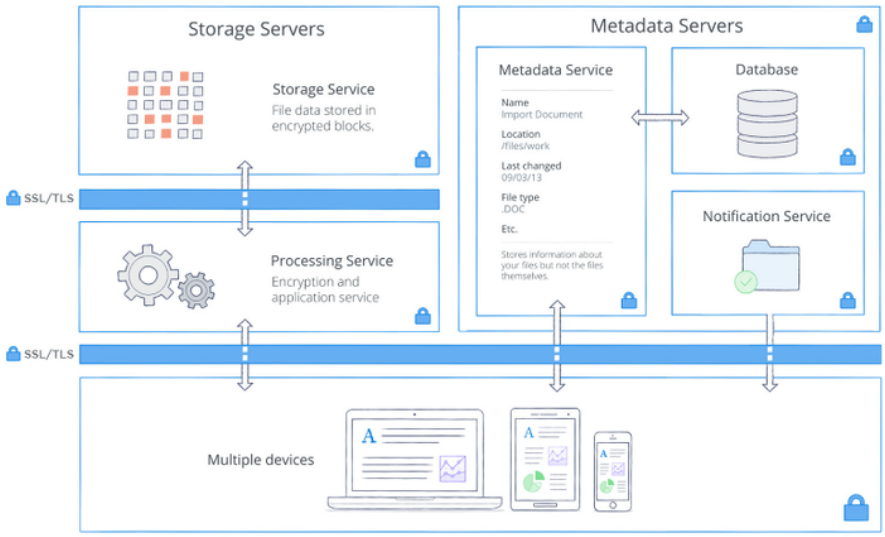
5.2 Dropbox

Dropbox [43] je ponudnik storitve hranjenja podatkov v oblaku, velikokrat namenjen izmenjavi datotek. Dropbox je dostopen na Windows, Macintosh in Linux operacijskih sistemih.

Storitev nam ponuja 2 GB prostora pri brezplačnem načinu uporabe, katerega pa lahko z dodatno članarino še razširimo. Uporabniki lahko dostopajo do storitve preko namenske aplikacije ali pa kar preko spletnega brskalnika. Tako

se mora uporabnik le prijaviti s svojim računom in pridobi pregled nad svojimi datotekami kjerkoli kjer je internetna povezava.

Kot je prikazano na sliki 5.1, lahko uporabniki dodajajo, brišejo ali pa si delijo datoteke. Datoteke so lahko poljubnih vrst, kot so dokumenti, fotografije in video. Vse spremembe bodo avtomatično vidne na vseh napravah (osebni računalniki, telefoni in preko spletne strani), kar nam omogoča dostop od koderkoli, obenem pa imamo zagotovljeno varnost nad našimi podatki. Na sliki je predstavljeno interno delovanje strežnika, njegove komponente in uporabniški vmesnik. Razvidno je, da je podatkovni strežnik ločen od informacijskega, ki služi z namenom, da izvemo o podatkih kakšno informacijo več, kot je na primer tip datoteke, kdaj je bila nazadnje spremenjena, njeno lokacijo v datotečnem sistemu ustvarjenem v oblaku in ostalo. Hkrati je razvidno, da je servis, ki služi za opomnike ločen od ostalih informacij o podatkih. Namenjen je sinhronizaciji uporabniških naprav, saj uporabnika spomni, da je v oblaku na voljo nova oziroma spremenjena datoteka. Vidimo, da je komunikacija iz opomniške storitve enosmerna, saj iz nje zgolj prejemamo informacije o spremembah. Druga stran oblačne storitve Dropbox je podatkovni strežnik, ki hrani vse podatke, ki jih uporabniki imajo. Ob naložitvi podatkov se ti najprej obdelajo, da jih strežnik lažje shrani, ko pa podatke beremo, pa se ti spremenijo nazaj v osnovno naloženo obliko. Do obeh strani imajo možnost dostopa vse naprave, ki podpirajo storitev.



Slika 5.1: Abstrakcija delovanja oblačne storitve.

5.3 Izbira tehnologij

Pri testiranju smo se odločili uporabiti tehnologije, ki so opisane v nadaljevanju.

5.3.1 BASH

Bash [44] je Unix ukazna lupina (ang. *shell*). To je program, katerega namen je omogočiti komunikacijo med uporabnikom in jedrom operacijskega sistema. Komunikacija poteka znakovno, kar pomeni, da uporabnik preko tipkovnice vnaša ukaze, na katere odziv sistema lahko prebere z zaslona.

V ukazni lupini poganjamo programe, s katerimi izvajamo željena opravila. Torej interpretira ukaze, ki jih vpisujemo preko tipkovnice in izvaja programe, ki jih kot uporabnik zahtevamo. Opravila, ki jih zahtevamo, se vejijo vse od uporabniških programov, do najrazličnejših administrativnih opravil in nastavitev sistema. Ob izvajanju programov lupina oblikuje zapise in koordinira izvajanje paralelnih zahtevkov. Vse izhaja iz zamisli, da delo razdelimo na manjše in točno določene naloge, katere učinkovito razrešijo namenski programi. Naloga uporabnika tako postane, da preko ukazne lupine poganja programe v pravilnem vrstnem redu.

Prednost takega pristopa je, da imamo tako na voljo množico preizkušenih programov, katere pa moramo le pravilno povezati v skripti kot neko celoto, ki rešuje določen problem.

5.3.2 curl

curl [45] je orodje, dostopno kot program v ukazni lupini bash, ki je namenjeno prenosu podatkov na, ali s strežnikov, z uporabo raznih podprtih protokolov (DICT, FILE, FTP, FTPS, GOPHER, HTTP, HTTPS, IMAP, IMAPS, LDAP, LDAPS, POP3, POP3S, RTMP, RTSP, SCP, SFTP, SMB, SMBS, SMTP, SMTPS, TELNET in TFTP). curl lahko tako uporabimo za FTP prenos, HTTP zahteve, SSL povezave, piškote in tudi uporabniško avtentikacijo.

5.3.3 Dropbox API

Pri meritvah se sklicujemo na Dropbox API [46], kar nam omogoča neposredno komunikacijo s programskimi komponentami Dropboxa. Tako s pomočjo množice HTTP končnih točk lahko integriramo željeno aplikacijo. Dropbox ponuja lastno HTTP dokumentacijo za API, ki ponuja vse kar lahko z njim storimo. Nudi tudi organizirane odgovore na pogosta vprašanja glede Dropbox API in ponuja primerke uporabe.

5.4 Definicija bremena storitve

Z uporabo napisane skripte bomo testirali scenarije bremen zmogljivosti opisane v nadaljevanju. Ker v raznih scenarijih potrebujemo kopico različnih velikosti

datotek smo se odločili, da bomo datoteke, potrebne pri izvajanju testov, avtomatizirano generirali. To nam dovoljuje tudi dejstvo, da so datoteke na disku, neglede na njihovo vsebino, enakovredne (binarne) in nimamo pristranosti kar se tiče tipa teh datotek.

5.4.1 Latenca

ping [47] je administrativno orodje (program), ki ga ponavadi uporabljamo za test dosegljivosti (ang. *reachability*) gostitelja na internetnem protokolu (IP) omrežja. Izmeri čas, ki ga potrebujemo do destinacije in nazaj do vira, od koder smo poslali sporočilo. Ta čas imenujemu tudi round-trip time (RTT).

ping deluje tako, da pošilja t.i. Internet Control Message Protocol (ICMP) Echo Request pakete do ciljnega gostitelja in čaka na ICMP Echo Reply. ping nam tako lahko pomaga pri določitvi morebitnih napak, izgubi paketov na povezavi, obvesti nas pa tudi s statističnim povzetkom o povprečju, standardnem odklonu, ter minimalnem in maksimalnem času RTT.
Z izvedbo ukaza ping preko ukazne lupine ali skripte določimo čas dostopa.

Izmerjeni povprečni časi dostopa za US-West območje (California - 108.160.172.65) so sledeči:

- rtt minimum 178.815 ms
- rtt povprečje 179.935 ms
- rtt maksimum 184.236 ms
- rtt std. deviacija 1.537 ms

10 paketov poslanih (84 bajtov), 10 prejetih (64 bajtov), 0% izguba paketov, čas 9012 ms.

Izmerjeni povprečni časi dostopa za US-East območje (Virginia - 45.58.74.1) so sledeči:

- rtt minimum 112.980 ms
- rtt povprečje 118.066 ms
- rtt maksimum 144.773 ms
- rtt std. deviacija 9.048 ms

10 paketov poslanih (84 bajtov), 10 prejetih (64 bajtov), 0% izguba paketov, čas 9011 ms.

Izmerjeni povprečni časi dostopa za EU območje (Germany - 162.125.66.7) so sledeči:

- rtt minimum 26.090 ms
- rtt povprečje 27.257 ms
- rtt maksimum 29.141 ms
- rtt std. deviacija 0.933 ms

10 paketov poslanih (84 bajtov), 10 prejetih (64 bajtov), 0% izguba paketov, čas 9015 ms.

Izmerjeni povprečni časi dostopa za Asia območje (Japan - 162.125.80.1) so sledeči:

- rtt minimum 273.684 ms
- rtt povprečje 299.178 ms
- rtt maksimum 351.497 ms
- rtt std. deviacija 25.935 ms

10 paketov poslanih (84 bajtov), 10 prejetih (64 bajtov), 0% izguba paketov, čas 9009 ms.

5.4.2 Prenos posameznih datotek

Z uporabo skripte 5.2 za kreiranje datotek pri testiranju generiramo velikosti datotek 1B, 10B, 100B, 1KiB, 10KiB, ,100KiB, 1MiB, 10MiB. Posamezna datoteka je avtomatsko narejena z uporabo skripte za kreiranje datotek. Te datoteke uporabimo kot breme pri testiranju prenosov na in s strežnika oblačne storitve, kjer bomo merili čas, ki ga potrebujemo za prenos in hitrost pri kateri se prenos vrši.

5.4.3 Prenos več datotek skupaj z nadzirano pogostostjo

Pri tem bremenu skušamo ugotoviti, kako se spreminja odzivnost strežnika, ko povečujemo frekvenco zahtevkov. Tako skušamo ugotoviti, kje se nahaja točka nasičenja. Izberemo določeno velikost bremena, katerega frekvenco prejemanja ali oddajanja kontrolirano nadzorujemo, obenem pa spremljamo odzivnost obremenjenega strežnika.

5.5 Definicija metrik in orodij za meritve

V nadaljevanju opišemo definicije metrik in orodij za meritve.

5.5.1 Čas

Pri merjenju časa T se osredotočamo na čas, ki je potreben za celoten proces komunikacije. Čas T je v osnovi definiran kot:

$$T = T_1 + T_2 + T_3.$$

Tu posamezni členi vsote predstavljajo čas potovanja sporočila do prejemnika, čas obdelave zahteve in čas potovanja sporočila do naslovnika. S to metriko lahko torej izmerimo trajanje celotne seje pri določeni zahtevi. To nam pride v pomoč ob ocenjevanju potrebnega časa za strežbo določene zahteve.

5.5.2 Prepustnost

Pri prepustnosti opazujemo obdelovanje zahtev na strežniku ob spreminjanju intenzivnost zahtevkov. Za to metriko pričakujemo, da se bo sprva sorazmerno povečevala z večanjem intezivnosti prihajanja zahtev, na določeni točki pa pričakujemo uklon, kar nam namiguje na preobremenjenost sistema. S pomočjo te metrike skušamo oceniti kdaj nastopi preobremenjenost sistema in kaj se s sistemom dogaja ob spreminjanju intenzivnosti zahtev.

5.5.3 BASH skripta za izvajanje testov

Skripta 5.1 je namenjena izvajanju testov hitrosti obdelave različnih velikosti datotek. Zajema glavni akciji za prejemanje in oddajanje zahtev, omogoča pa tudi nadzor nad frekvenco zahtevkov, njihovo velikostjo in trajanjem.

Listing 5.1: Skripta za izvajanje testov.

```
#!/bin/bash
#script: run_test.sh

#Definicije konstant in spremenljivk
FILE_SIZE_ARRAY=("1" "10" "100" "1KiB" "10KiB" "100KiB" "1MiB" "10MiB")
unset FILE_CREATED_ARRAY
unset COUNT
unset SIZE
unset DELAY
unset WINDOW

FILE_PREFIX="bin_file"
FILE_UPLOAD_PREFIX="up/"$FILE_PREFIX
FILE_UPLOAD_SUFFIX="up"
FILE_DOWNLOAD_PREFIX="down/"$FILE_PREFIX
FILE_DOWNLOAD_SUFFIX="down"

SILENT=0
SHOW_PROGRESSBAR=0


#Funkcija za pomoc
function f_help {
   echo "Usage: ./$(basename $0) [MODE(optional)] [CONTROL] [ACTION]"
   echo ""
   echo "Modes: "
   echo "-s silent"
   echo "-p show progressbar"
   echo ""
   echo "Control: "
   echo "-c count"
   echo "-z size in bytes"
   echo "-d delay in secondds"
   echo "-w window size"
   echo "-t duration time"
   echo ""
   echo "Actions: [up | down | up-parallel | down-parallel | help]"
   echo ""
   echo "Actions: [up-parallel | down-parallel] require control -c -z
       -d/w"
}
```

```
#Funkcija za prenos parametrov
function f_get_parameters {
   #Preverimo kater parameter prenasamo
   if [[ $SHOW_PROGRESSBAR == 1 && $SILENT == 0 ]]; then
      CARRY_PARAMETERS="-p"
   elif [[ $SHOW_PROGRESSBAR == 0 && $SILENT == 0 ]]; then
      CARRY_PARAMETERS=""
   else
      CARRY_PARAMETERS="-s"
   fi

   echo $CARRY_PARAMETERS
}

#Funkcija za kreiranje datotek preko polja
function f_create_files {
   echo "Creating files: "
   tLen=${#FILE_SIZE_ARRAY[@]}
   #Za vsak element polja kreiramo datoteko
   for (( i=0; i<${tLen}; i++ )); do
      echo -n "Creating file: [$FILE_PREFIX${FILE_SIZE_ARRAY[$i]}]..."

      #Klic skripte za alociranje datotek
      message=$(bash allocate_file.sh
          "$FILE_PREFIX${FILE_SIZE_ARRAY[$i]}" "${FILE_SIZE_ARRAY[$i]}")

      #Preverimo izhodni status
      if [[ $? == 0 ]]; then
         echo "DONE"
         FILE_CREATED_ARRAY[${#FILE_CREATED_ARRAY[@]}]="${FILE_SIZE_ARRAY[$i]}"
      else
         echo "FAIL"
         echo $message
      fi
   done
   echo -e "\n"
}

#Funkcija za brisanje datotek
function f_delete_files {
   echo "Deleting files:"
   #Za vsak element polja zbrisemo datoteko
   for (( i=0; i<${#FILE_CREATED_ARRAY[@]}; i++ )); do
      echo -n "Deleting file: [$FILE_PREFIX${FILE_CREATED_ARRAY[$i]}]..."
      rm "$FILE_PREFIX${FILE_CREATED_ARRAY[$i]}"

      #Preverimo izhodni status
      if [[ $? == 0 ]]; then
         echo "DONE"
      else
```

```
        echo "FAIL"
      fi
   done
}

#Funkcija za opredelitev statistik
function f_read_summary_files {
   dir=$1
   count=$2

   #Glede na spremenljivko $SILENT priredimo izpis
   if [[ $SILENT == 0 ]]; then
      for file in $dir; do
        cat "$file"
        echo -e "\n"
      done
   else
      for file in $dir; do
        cat "$file" | grep "RETRY\|FAILED\|Retry time"
      done
   fi

   echo "SUMMARY (average for one file): "

   #Izracun poprecnih statistik za posamezno datoteko
   grep -h "Lookup time:.*\|Connect time:.*\|PreXfer time:.*\|StartXfer
       time:.*\|Total time:.*\|Download speed:.*\|Upload speed:.*\|Retry
       time:.*" $dir | awk -F '\t' -v count="$count"
       '{sum[$1]=sum[$1]+$2} END {for (a in sum) {print
       a"\t"sum[a]/count};print "Full time:\t"(sum["Total
       time:"]+sum["Retry time:"])/count}'

   echo -e "\n"
}

#Funkcija za zaporedno oddajanje datotek
function f_test_up {
   local DURATION=$1
   if [ -z "$DURATION" ]; then
      DURATION=0
   fi
   local END=$(( SECONDS + $DURATION ))
   local CARRY_PARAMETERS=$(f_get_parameters)

   #Dokler nam cas dovoljuje, izvajamo prenose
   while [ $SECONDS -lt $END ] || [ $DURATION -eq 0 ]; do
      for (( i=0; i<${#FILE_CREATED_ARRAY[@]}; i++ )); do
         #Preverimo cas namenjen izvajanju
         if [ $SECONDS -gt $END ] && [ $DURATION -ne 0 ]; then
            break
```

```
        fi
        #Klic skripte za oddajanje in prejemanje datotek
        bash dropbox_ud.sh $CARRY_PARAMETERS "upload"
            "$FILE_PREFIX${FILE_CREATED_ARRAY[$i]}"
            "$FILE_UPLOAD_PREFIX${FILE_CREATED_ARRAY[$i]}$FILE_UPLOAD_SUFFIX"
        echo -e "\n"
      done

      #Preverimo cas namenjen izvajanju
      if [ $SECONDS -ge $END ]; then
        break
      fi
    done
}

#Funkcija za zaporedno prejemanje datotek
function f_test_down {
    local DURATION=$1
    if [ -z "$DURATION" ]; then
      DURATION=0
    fi
    local END=$(( SECONDS + $DURATION ))
    local CARRY_PARAMETERS=$(f_get_parameters)

    #Dokler nam cas dovoljuje, izvajamo prenose
    while [ $SECONDS -le $END ] || [ $DURATION -eq 0 ]; do
      for (( i=0; i<${#FILE_CREATED_ARRAY[@]}; i++ )); do
        #Preverimo cas namenjen izvajanju
        if [ $SECONDS -gt $END ] && [ $DURATION -ne 0 ]; then
          break
        fi
        #Klic skripte za oddajanje in prejemanje datotek
        bash dropbox_ud.sh $CARRY_PARAMETERS "download"
            "$FILE_UPLOAD_PREFIX${FILE_CREATED_ARRAY[$i]}$FILE_UPLOAD_SUFFIX"
            "$FILE_DOWNLOAD_PREFIX${FILE_CREATED_ARRAY[$i]}$FILE_DOWNLOAD_SUFFIX"
        echo -e "\n"
      done

      #Preverimo cas namenjen izvajanju
      if [ $SECONDS -ge $END ]; then
        break
      fi
    done
}

#Funkcija za simultano oddajanje datotek
function f_test_up_parallel {
    local COUNT=$1
    local SIZE=$2
    local DELAY=$3
```

```
  local WINDOW=$4
  local DURATION=$5
  if [ -z "$DURATION" ]; then
     DURATION=0
  fi
  local END=$(( SECONDS + $DURATION ))
  local CARRY_PARAMETERS=$(f_get_parameters)

  #Dokler nam cas dovoljuje, izvajamo prenose
  iter=1
  while [ $SECONDS -le $END ] || [ $DURATION -eq 0 ]; do
     for (( i=$iter; i<=$COUNT; i++ )); do
        #Preverimo cas namenjen izvajanju
        if [ $SECONDS -gt $END ] && [ $DURATION -ne 0 ]; then
           COUNT=$(( $i - 1 ))
           break
        fi

        #Klic skripte za oddajanje in prejemanje datotek
        $(bash dropbox_ud.sh $CARRY_PARAMETERS "upload"
            "$FILE_PREFIX${FILE_CREATED_ARRAY[0]}"
            "$FILE_UPLOAD_PREFIX${FILE_CREATED_ARRAY[0]}$FILE_UPLOAD_SUFFIX$i"
            > "log/log$FILE_UPLOAD_SUFFIX$i.txt") &

        #Preverimo ali delujemo v nacinu okna ali zakasnitve
        if [ -z "$WINDOW" ]; then
           sleep $DELAY
        else
           WINDOW=$(( $WINDOW - 1 ))

           #Preverimo, ce je okno se prosto
           if [ "$WINDOW" -gt 0 ]; then
              continue
           else
              wait -n
              WINDOW=$(( $WINDOW + 1 ))
           fi
        fi
     done
     #Preverimo cas namenjen izvajanju
     if [ $SECONDS -ge $END ]; then
        break
     fi
     tempiter=$iter
     iter=$(( $COUNT + 1 ))
     COUNT=$(( $COUNT + $COUNT - $tempiter + 1 ))
  done
  wait

  #Preberemo statisticni povzetek
```

```
   f_read_summary_files "$(eval echo
       "log/log$FILE_UPLOAD_SUFFIX"{1..$COUNT}".txt")" $COUNT
}

#Funkcija za simultano prejemanje datotek
function f_test_down_parallel {
   local COUNT=$1
   local SIZE=$2
   local DELAY=$3
   local WINDOW=$4
   local DURATION=$5
   if [ -z "$DURATION" ]; then
      DURATION=0
   fi
   local END=$(( SECONDS + $DURATION ))
   local CARRY_PARAMETERS=$(f_get_parameters)

   #Dokler nam cas dovoljuje, izvajamo prenose
   iter=1
   while [ $SECONDS -le $END ] || [ $DURATION -eq 0 ]; do
      for (( i=$iter; i<=$COUNT; i++ )); do
         #Preverimo cas namenjen izvajanju
         if [ $SECONDS -gt $END ] && [ $DURATION -ne 0 ]; then
            COUNT=$(( $i - 1 ))
            break
         fi

         #Klic skripte za oddajanje in prejemanje datotek
         $(bash dropbox_ud.sh $CARRY_PARAMETERS "download"
             "$FILE_UPLOAD_PREFIX${FILE_CREATED_ARRAY[0]}$FILE_UPLOAD_SUFFIX$i"
             "$FILE_DOWNLOAD_PREFIX${FILE_CREATED_ARRAY[0]}$FILE_DOWNLOAD_SUFFIX$i"
             > "log/log$FILE_DOWNLOAD_SUFFIX$i.txt") &

         #Preverimo ali delujemo v nacinu okna ali zakasnitve
         if [ -z "$WINDOW" ]; then
            sleep $DELAY
         else
            WINDOW=$(( $WINDOW - 1 ))

            #Preverimo, ce je okno se prosto
            if [ "$WINDOW" -gt 0 ]; then
               continue
            else
               wait -n
               WINDOW=$(( $WINDOW + 1 ))
            fi
         fi
      done
      #Preverimo cas namenjen izvajanju
      if [ $SECONDS -ge $END ]; then
```

```
        break
      fi
      tempiter=$iter
      iter=$(( $COUNT + 1 ))
      COUNT=$(( $COUNT + $COUNT - $tempiter + 1 ))
   done
   wait

   #Preberemo statisticni povzetek
   f_read_summary_files "$(eval echo
       "log/log$FILE_DOWNLOAD_SUFFIX"{1..$COUNT}".txt")" $COUNT
}


#####MAIN#####
#Opredelitev prametrov
while getopts ":c:z:d:w:t:ps" opt; do
    case $opt in

   c)
      COUNT=$OPTARG
      ;;
   z)
      SIZE=$OPTARG
      ;;
   d)
      DELAY=$OPTARG
      ;;
   w)
      WINDOW=$OPTARG
      ;;
   t)
      DURATION=$OPTARG
      ;;
   p)
      SHOW_PROGRESSBAR=1
         ;;
    s)
         SILENT=1
         ;;
    \?)
         echo "Invalid option: -$OPTARG" >&2
         exit 1
         ;;
    :)
         echo "Option -$OPTARG requires an argument." >&2
         exit 1
         ;;

   esac
```

```
done
shift $(( OPTIND - 1 ))

#Prenos vhodnih argumentov
ACTION="$1"

#Odlocitev glede na akcijo
case $ACTION in
   (up)
      #Zaporedno oddajanje datotek
      f_create_files
      f_test_up "$DURATION"
      f_delete_files
   ;;
   (down)
      #Zaporedno prejemanje datoek
      f_create_files
      f_test_down "$DURATION"
      f_delete_files
   ;;
   (up-parallel)
      #Simultano oddajanje datoek
      #Preverimo podane parametre
      if [ -z "$COUNT" ] || [ -z "$SIZE" ]; then
         echo "Wrong action!"
         f_help
         exit 1
      fi
      if [ -z "$WINDOW" ]; then
         if [ -z "$DELAY" ]; then
            echo "Wrong action!"
            f_help
            exit 1
         fi
      fi

      FILE_SIZE_ARRAY=("$SIZE")
      f_create_files
      f_test_up_parallel "$COUNT" "$SIZE" "$DELAY" "$WINDOW" "$DURATION"
      f_delete_files
   ;;
   (down-parallel)
      #Simultano prejemanje datotek
      #Preverimo podane parametre
      if [ -z "$COUNT" ] || [ -z "$SIZE" ]; then
         echo "Wrong action!"
         f_help
         exit 1
      fi
      if [ -z "$WINDOW" ]; then
```

```
        if [ -z "$DELAY" ]; then
          echo "Wrong action!"
          f_help
          exit 1
        fi
      fi

      FILE_SIZE_ARRAY=("$SIZE")
      f_create_files
      f_test_down_parallel "$COUNT" "$SIZE" "$DELAY" "$WINDOW"
          "$DURATION"
      f_delete_files
    ;;
    (help)
      #Pomoc
      f_help
    ;;
    (*)
      echo "Wrong action!"
      f_help
      exit 1
esac

exit 0
```

5.5.4 BASH skripta za kreiranje datotek

Skripta 5.2 je namjena natančni opredelitvi datotek, ki jih bomo uporabili pri našem testiranju. Omogoča avtomatizirano generiranje poljubne velikosti datoteke preko sistemskega klica fallocate [48], ki nam prealocira prostor za posamezno datoteko.

Listing 5.2: Skripta za kreiranje datotek.

```
#!/bin/bash
#script: allocate_file.sh

#Funkcija za pomoc
function f_help {
   echo "Usage: ./$(basename $0) [FILENAME] [SIZE]"
}

#Funkcija za kreiranje datoteke
function f_allocate_file {
   local FILE_NAME=$1
   local FILE_SIZE=$2

   #Preverimo ime in velikost
   if [ -z "$FILE_NAME" ] || [ -z "$FILE_SIZE" ]; then
```

```
      echo "Wrong action!"
      f_help
      exit 1
   fi

   #Preverimo, ce datoteka s tem imenom ze obstaja
   if [ -e "$FILE_NAME" ]; then
      echo "File ["$FILE_NAME"] allready exists!"
      exit 1;
   else
      #Sistemski klic ki nam alocira datoteko
      fallocate -l "$FILE_SIZE" "$FILE_NAME"
   fi
}


#####MAIN#####
#Prenos vhodnih argumentov
FILE_NAME=$1
FILE_SIZE=$2

#Kreiramo datoteko z zeljenim imenom in velikostjo
f_allocate_file $FILE_NAME $FILE_SIZE

exit 0
```

5.5.5 BASH skripta za prenos datotek

Skripta 5.3 je namenjena prenosu datototek. Uporabljen pa je bil vmesnik Dropbox API [46] za povezavo s strežnikom in curl [45] za prenos podatkov preko mreže.

Listing 5.3: skripta za prenos datotek.

```
#!/bin/bash
#script: dropbox_ud.sh

#Definicije konstant in spremenljivk
API_UPLOAD_URL="https://content.dropboxapi.com/2/files/upload"
API_DOWNLOAD_URL="https://content.dropboxapi.com/2/files/download"
OAUTH_ACCESS_TOKEN="<TOKEN>"
SILENT=0
SHOW_PROGRESSBAR=0


#Funkcija za pomoc
function f_help {
   echo "Usage: ./$(basename $0) [MODE(optional)] [ACTION] [SOURCE]
       [DESTINATION]"
```

```
   echo ""
   echo "Modes: "
   echo "-s silent"
   echo "-p show progressbar"
   echo ""
   echo "Actions: [upload | download | help]"
}

#Funkcija za izpis glede na $SILENT spremenljivko
function print {
    if [[ $SILENT == 0 ]]; then
       echo -ne "$1";
    fi
}

#Funkcija za oddajanje
function f_upload {
   local FILE_SOURCE="$1"
   local FILE_DESTINATION="$2"
   RESPONSE_FILE="response/"$(basename $FILE_DESTINATION)"response.txt"

   #Normaliziramo pot
   if [[ $FILE_DESTINATION != /* ]]; then
      FILE_DESTINATION="/"$FILE_DESTINATION
   fi

   #Glede na parametre dolocimo nacin izpisa
   if [[ $SHOW_PROGRESSBAR == 1 && $SILENT == 0 ]]; then
        CURL_PARAMETERS="--progress-bar"
        LINE_CR="\n"
   elif [[ $SHOW_PROGRESSBAR == 0 && $SILENT == 0 ]]; then
      CURL_PARAMETERS=""
      LINE_CR="\n"
   else
      CURL_PARAMETERS="-L -s"
      LINE_CR=""
   fi

   print "> Uploading \"$FILE_SOURCE\" to \"$FILE_DESTINATION\"...
       $LINE_CR"

   #Izvedemo prenost in izpisemo statistike
   curl -X POST $API_UPLOAD_URL \
   --header "Authorization: Bearer $OAUTH_ACCESS_TOKEN" \
   --header "Dropbox-API-Arg: {\"path\": \"$FILE_DESTINATION\"}" \
   --header "Content-Type: application/octet-stream" \
   --data-binary @"$FILE_SOURCE" \
   $CURL_PARAMETERS -D "$RESPONSE_FILE" -o /dev/null \
   -w 'Lookup time:\t%{time_namelookup}\nConnect
       time:\t%{time_connect}\nPreXfer
```

```
      time:\t%{time_pretransfer}\nStartXfer
      time:\t%{time_starttransfer}\nTotal
      time:\t%{time_total}\nDownload speed:\t%{speed_download}\nUpload
      speed:\t%{speed_upload}\n'

  #Preverimo odgovor streznika
  CURL_STATUS=$?
  if grep -q "^HTTP/1.1 200 OK" "$RESPONSE_FILE"; then
     print "DONE\n"
  else
     #Preverimo tip napake
     if grep -q "^HTTP/1.1 429 too_many_write_operations/"
         "$RESPONSE_FILE"; then
        echo -e "Retry time:\t"$( grep "Retry-After:" "$RESPONSE_FILE"
            | cut -f2 -d " " | tr -dc '0-9')"\n"
        sleep $( grep "Retry-After:" "$RESPONSE_FILE" | cut -f2 -d " "
            | tr -dc '0-9')
        echo -e "RETRY: HTTP/1.1 429 too_many_write_operations/, CURL
            status: $CURL_STATUS\n"
        f_upload "$1" "$2"
     else
        echo -e "FAILED: $(head -n 1 $RESPONSE_FILE), CURL status:
            $CURL_STATUS\n"
        f_upload "$1" "$2"
     fi
  fi
}

#Funkcija za prejemanje
function f_download {
  local FILE_SOURCE="$1"
  local FILE_DESTINATION="$2"
  RESPONSE_FILE="response/"$(basename $FILE_DESTINATION)"response.txt"

  #Normaliziramo pot
  if [[ $FILE_SOURCE != /* ]]; then
     FILE_SOURCE="/"$FILE_SOURCE
  fi

  if [ -e "$FILE_DESTINATION" ]; then
     print "File ["$FILE_DESTINATION"] allready exists!"
     exit 1;
  fi

  #Glede na parametre dolocimo nacin izpisa
  if [[ $SHOW_PROGRESSBAR == 1 && $SILENT == 0 ]]; then
        CURL_PARAMETERS="--progress-bar"
        LINE_CR="\n"
     elif [[ $SHOW_PROGRESSBAR == 0 && $SILENT == 0 ]]; then
        CURL_PARAMETERS=""
```

```
        LINE_CR="\n"
    else
        CURL_PARAMETERS="-L -s"
        LINE_CR=""
    fi

  print "> Downloading \"$FILE_SOURCE\" to \"$FILE_DESTINATION\"...
      $LINE_CR"

  #Izvedemo prenost in izpisemo statistike
  curl -X POST $API_DOWNLOAD_URL \
  --header "Authorization: Bearer $OAUTH_ACCESS_TOKEN" \
  --header "Dropbox-API-Arg: {\"path\": \"$FILE_SOURCE\"}" \
  -o "$FILE_DESTINATION" \
  $CURL_PARAMETERS -D "$RESPONSE_FILE" \
  -w 'Lookup time:\t%{time_namelookup}\nConnect
      time:\t%{time_connect}\nPreXfer
      time:\t%{time_pretransfer}\nStartXfer
      time:\t%{time_starttransfer}\nTotal
      time:\t%{time_total}\nDownload speed:\t%{speed_download}\nUpload
      speed:\t%{speed_upload}\n'

  #Preverimo odgovor streznika
  CURL_STATUS=$?
  if grep -q "^HTTP/1.1 200 OK" "$RESPONSE_FILE"; then
     print "DONE\n"
     echo -e "PID:\t"$BASHPID
     rm -fr "$FILE_DESTINATION"
  else
     #Preverimo tip napake
     if grep -q "^HTTP/1.1 429 too_many_requests/" "$RESPONSE_FILE";
         then
        echo -e "Retry time:\t"$( grep "Retry-After:" "$RESPONSE_FILE"
            | cut -f2 -d " " | tr -dc '0-9')"\n"
           sleep $( grep "Retry-After:" "$RESPONSE_FILE" | cut -f2 -d "
               " | tr -dc '0-9')
           echo -e "RETRY: HTTP/1.1 429 too_many_requests/, CURL status:
               $CURL_STATUS\n"
           f_download "$1" "$2"
        else
        echo -e "FAILED: $(head -n 1 $RESPONSE_FILE), CURL status:
            $CURL_STATUS\n"
        rm -fr "$FILE_DESTINATION"
        f_download "$1" "$2"
        fi
  fi
}


#####MAIN#####
```

```
#Opredelitev prametrov
while getopts ":sp" opt; do
    case $opt in

    s)
      SILENT=1
    ;;

    p)
      SHOW_PROGRESSBAR=1
    ;;

    \?)
      echo "Invalid option: -$OPTARG" >&2
      exit 1
    ;;

  esac
done
shift $(( OPTIND - 1 ))

#Prenos vhodnih argumentov
ACTION="$1"
FILE_SOURCE="$2"
FILE_DESTINATION="$3"

#Odlocitev glede na akcijo
case $ACTION in
   (upload)
      #Oddajanje
      f_upload "$FILE_SOURCE" "$FILE_DESTINATION"
   ;;
   (download)
      #Prejemanje
      f_download "$FILE_SOURCE" "$FILE_DESTINATION"
   ;;
   (help)
      #Pomoc
      f_help
   ;;
   (*)
      echo "Wrong action!"
      f_help
      exit 1
esac

exit 0
```

5.6 Rezultati meritev

V nadaljevanju sledijo rezultati meritev, ki smo jih opravili v okviru našega dela.

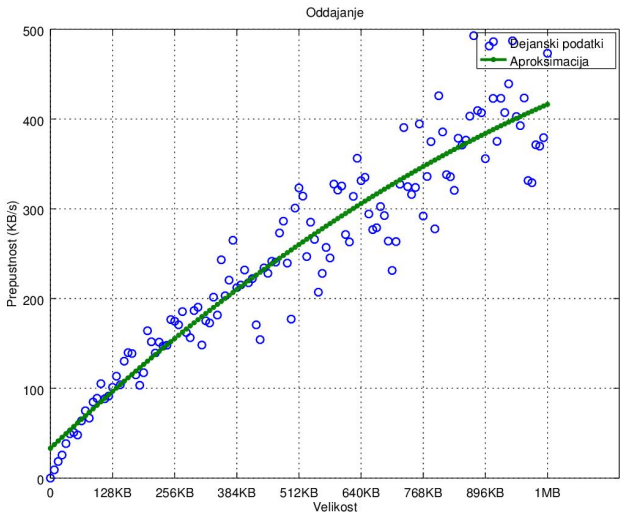
5.6.1 Prenos posameznih datotek na strežnik

Meritev predstavlja prepustnost sistema pri oddajanju, ki smo jo dosegli ob prenosu različnih velikosti datotek na strežnik.

- **Hipoteza:** Naša hipoteza oziroma predpostavka je, da bo večinoma prepustnost naraščala linearno, ter da bo trajanje prenosa sorazmerno z velikostjo datoteke. Ob približevanju omejitvi hitosti naše internetne povezave pričakujemo, da se bodo pričenjale meritve uklanjati.

- **Robni pogoji:** To meritev smo opravljali tako, da smo linearno s konstantnim korakom (8192B) povečevali velikosti generiranih datotek. Temu je sledilo pošiljanje datotek tako, da smo eno za drugo poslali na strežnik, pri čemer smo za vsako datoteko počakali, da se je prenos uspešno zaključil. Za vsako datoteko smo izmerili povprečno hitrost z orodjem *curl*, pri kateri se je prenos vršil. Ta postopek merjenja smo sedemkrat ponovili. Meritev smo opravili enkrat na vsak dan v tednu, vsakič časovno neodvisno od prejšnjega merjenja. Na koncu smo izračunali povprečje hitrosti teh meritev, za vsako izmed velikosti datotek.

- **Komentar:** V grafu na sliki 5.2 opazimo podobnost logaritemski funkciji, kar nam napoveduje, da se približujemo maksimalni hitrosti prenosa, ki nam jo je predpisal ponudnik internetnih storitev. Pri večjih datotekah pa opazimo nekaj nihanja v hitrostih, kar si lahko razlagamo z raznimi nepredvidljivimi razmerami in anomalijami po omrežju. S to ugotovitvijo sklepamo tudi, da bodo smiselna bremena za naše meritve tja do 1MB. Namreč na meritve z večjim bremenom bi preveč vplivali z omejeno hitrostjo.



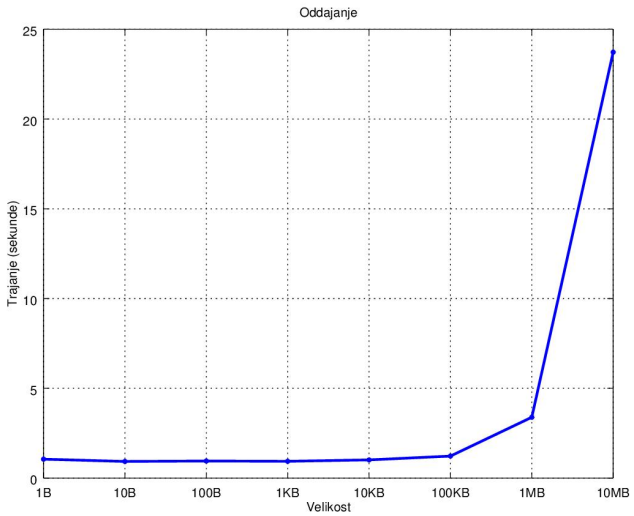
Slika 5.2: Prepustnost sistema pri oddajanju.

Za primerjavo smo izmerili tudi čas trajanja prenosa, katerega smo potrebovali pri prenosu posamezne datoteke na strežnik.

- **Hipoteza:** Predpostavka tu je, da bolj ko bodo datoteke velike, več časa bomo potrebovali za prenos. Pričakujemo tudi, da bo za večje datoteke čas bistveno večji, ker imamo omejeno hitrost oddajanja.

- **Robni pogoji:** Pri merjenju trajanja prenosa datotek na strežnik, smo se osredotočili na peščico datotek (1B, 10B, 100B, 1KB, 10KB, 100KB, 1MB, 10MB), katere smo temeljito obravnavali. Tudi tu smo pošiljali datotek tako, da smo pri vsaki datoteki počakali, da se je prenos uspešno zaključil, preden smo nadaljevali s pošiljanjem. Tu smo z orodjem *curl* merili celotne čase pri prenosu posameznih velikosti datotek. Za vsako izmed izbranih velikost datotek smo vsak dan, v časovnem obdobju enega tedna, naredili in povprečili deset meritev trajanja. Da bi še bolj zmanjšali pristranost, smo vsako meritev naredili časovno neodvisno od ostalih.

- **Komentar:** V grafu na sliki 5.3 je prikazan čas, ki smo ga potrebovali za prenos datotek na strežnik. Rezultat je povprečje večih in med seboj neodvisnih meritev. Velik skok pri meritvi zadnje datoteke, je posledica omejitve hitrosti ponudnika internetnih storitev.



Slika 5.3: Trajanje oddajanja.

5.6.2 Prenos posameznih datotek s strežnika

Meritve na sliki 5.4 predstavljajo kot rezultat prepustnost prenosa datotek s strežnika, torej pri prejemanju.

- **Hipoteza:** Tu smo pričakovali nekoliko bolj prevdarno linearnost med velikostjo in prepustnostjo, ter da bo trajanje prenosa sorazmerno z velikostjo datoteke. Ta pričakovanja so posledica dejstva, da se kasneje začnemo približevati omejitvi hitosti naše internetne povezave.

- **Robni pogoji:** Tudi tu smo pri meritvah linearno s konstantnim korakom (8192B) povečevali velikosti datotek. Datotekam smo eni za drugo z orodjem *curl* izmerili povprečno hitrost prenosa, pri čemer smo pri vsaki datoteki počakali, da se je prenos uspešno zaključil, preden smo nadaljevali s pošiljanjem. Tudi ta celoten postopek merjenja smo sedemkrat ponovili. Meritev smo opravljali enkrat na vsak dan v obdobju enega tedna. Merili smo ob nalkjučnih časih, vsakič neodvisno od prejšnjega merjenja. Tako smo na koncu lahko izračunali dober približek kot povprečje hitrosti pri vsaki izmed velikosti datotek.

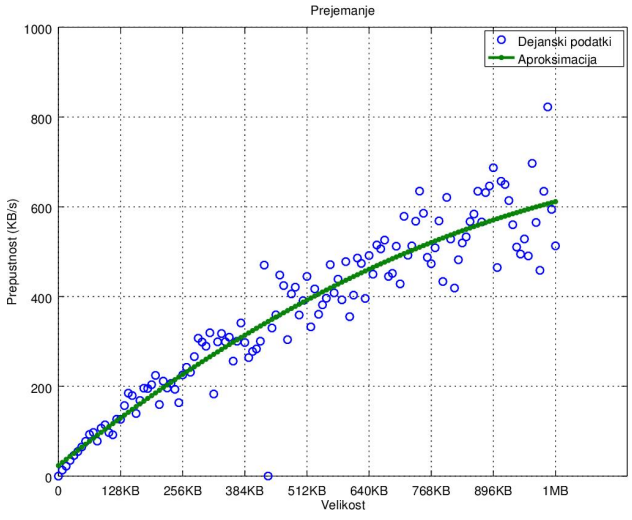
- **Komentar:** Iz grafa lahko zopet razberemo podobnost logaritemske funkcije, le da v primerjavi s prenosom na strežnik, kasneje začne limitirati, saj se kasneje začnemo približevati omejitvi hitrosti, predpisane s strani internetnega operaterja. Tudi tu opazimo vplive zunanjega omrežja, predvsem pri meritvah ki so trajale dlje časa in posledično bile dalj časa izpostavljene. Prav tako lahko tu sklepamo o primernih velikostih bremen do

približno 1MB, katera bodo primerna v prihodnjih testih, da ne bomo vplivali na rezultate zaradi omejene hitrosti.

Za razliko od eksperimenta, kjer spekter datotek pošiljamo na strežnik, tu opazino zanimiv sledeč pojav. Pri datoteki velikosti 432KB (442368B) opazimo nenavadno nizko prepustnost. Ob analizi prenosa v skripti ugotovimo, da nam strežnik odgovori z odgovorom 409 path/restricted_content/. Tako se prenos iregularno zaključi. Enak rezultat smo dobili, če smo spremenili ime te obravnavane datoteke. Prav tako tudi uporaba drugih orodij za alociranje datotek in drugačni načini alociranja niso privedli do sprememb. Prenos se je izvršil neuspešno le v primerih, ko smo orodju za alociranje datotek dovolili, da kreira datoteke z vsebino samih ničel. V preostalih kombinacijah vrednosti (npr. samih enic), ali pa če smo vsebino datoteke zakodirali kot tekst, vsebina ni postala zaščitena in prenos se je uspešno izvršil.

Ker vsebino samih ničel predstavljajo null terminatorji, bi to lahko begalo sistem na strani Dropbox strežnika in morda vidi to datoteko kot prazno. Podobne pojave so zasledili tudi drugi uporabniki pri drugih praznih datotekah, kar je omenjeno v virih [49] in [50].

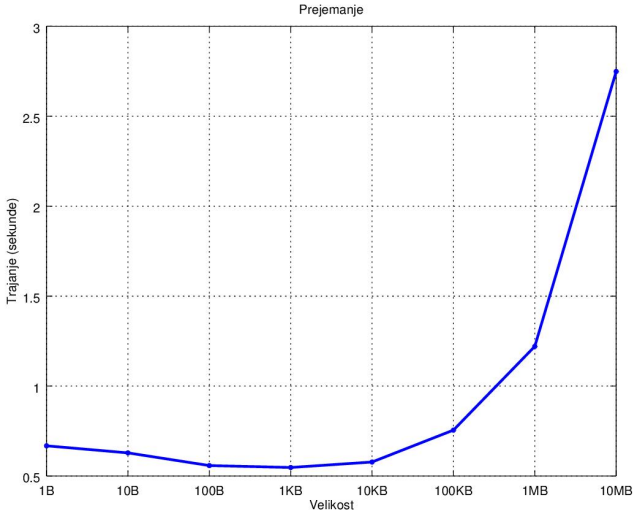
Izkaže se, da postane taka datoteka ob prenosu na strežnik zaščitena, in jo lahko prenesemo le preko spletnega brskalnika. Morda je razlog temu, velikost datoteke (442368B), ki je število sestavljeno iz zmnožka dveh različnih praštevil, ali pa gre le za nepolovljeno izjemo s strani Dropboxa.



Slika 5.4: Prepustnost sistema pri prejemanju.

Za primerjavo smo izmerili tudi čas trajanja prenosa, katerega smo potrebovali pri prenosu posamezne datoteke s strežnika.

- **Hipoteza:** Zopet predopstavljamo, da bolj ko bodo datoteke velike, več časa bomo potrebovali za prenos. Ker je naša maksimalna hitrost prenosa za prejemanje višja kot pri oddajanju pričakujemo, da bo prenos v tem primeru bistveno hitrejši.
- **Robni pogoji:** Pri merjenju trajanja prenosa datotek s strežnika, smo se prav tako osredotočili na peščico datotek (1B, 10B, 100B, 1KB, 10KB, 100KB, 1MB, 10MB), katere smo temeljito obravnavali. Pri prejemanju vsake datoteke smo počakali, da se je prenos uspešno zaključil, preden smo nadaljevali z zahtevo za prejemanje datoteke naslednje velikosti. Za vsako izmed izbranih velikost datotek, smo vsak dan v enem tednu naredili deset različnih meritev pri katerih smo izmerili čas trajanja. Čas prejemanja smo izmerili s pomočjo statističnega opisa, ki nam ga ponuja *curl*. Vsaka meritev je bila narejena časovno neodvisno od prejšjnih meritev ob naključnih časih. Te meritve smo na koncu še povprečili.
- **Komentar:** Na sliki 5.5 so prikazani časi, ki so bili potrebni za prenos datotek s strežnika. Rezultat je povprečje večih in med seboj neodvisnih meritev. Večji skok pri meritvi zadnje datoteke je posledica približevanju omejitvi hitrosti ponudnika internetnih storitev.

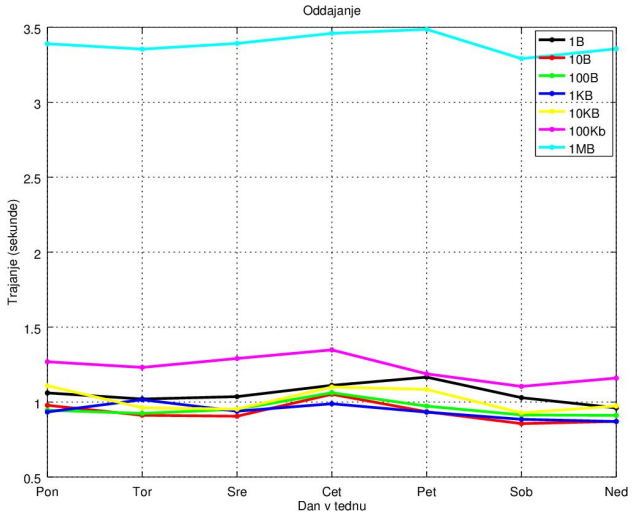


Slika 5.5: Trajanje prejemanja.

5.6.3 Prenos posameznih datotek skozi teden

Da bi ugotovili trend, kako se vrši prenos na strežnik med tednom, smo se odločili vsak dan izmeriti čas prenosa na strežnik.

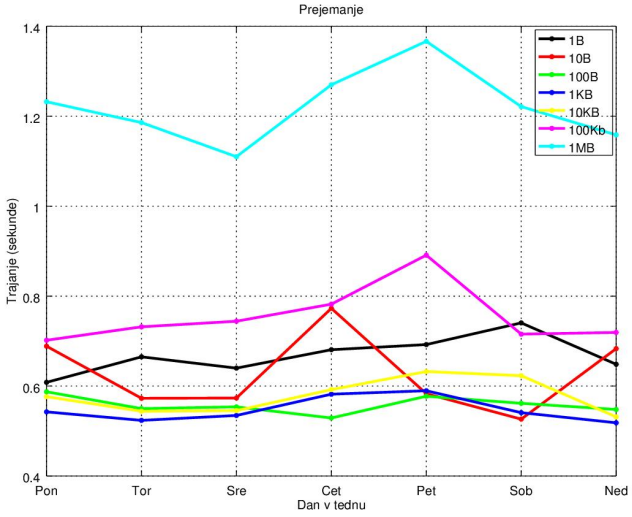
- **Hipoteza:** Naša hipoteza je, da se bodo časi prenosov za vse datoteke gibali približno podobno, s tem da bodo večje datoteke potrebovale več časa pri prenosu. Pričakujemo tudi, da bodo morda prenosi nekoliko daljši med tednom, kot pa v času vikenda.
- **Robni pogoji:** Meritev smo opravljali za vsako izmed obravnavanih velikosti datotek (1B, 10B, 100B, 1KB, 10KB, 100KB, 1MB), katerim smo pripisali graf faunkcije. Prenos smo izvedli tako, da smo ob vsakem oddajanju datoteke počakali, da se je prenos uspešno zaključil, preden smo nadaljevali s pošiljanjem datoteke naslednje velikosti. Pri vsaki izmed datotek smo vsak dan naredili petnajst različnih meritev, pri katerih smo izmerili čas trajanja s pomočjo *curl* statističnega opisa prenosa. Vsaka meritev je bila opravljena ob naključnih časih, nedvisno od prejšnje. Meritve smo tudi naredili neodvisno z dveh različnih lokacij in sicer deset meritev v okolici Ljubljane, preostalih pet pa v centru Ljubljanje. Na koncu smo izračunali še povprečni skupni čas obeh lokacij.
- **Komentar:** V grafu na sliki 5.6 opazimo da se vse različne velikosti približno enakovredne po trajanju prenosa, z razliko od datoteke velikost 1MB, ki nekoliko prevladuje nad ostalimi. Opazimo lahko tudi, da se trajanje prenosa nekoliko poveča v sredi tedna in na prehodu iz nedelje na ponedeljek.



Slika 5.6: Trajanje prenosa skozi teden pri oddajanju.

Podobno smo se odločili narediti pri prejemanju istih datotek s strežnika.

- **Hipoteza:** Pri prejemanju pričakujemo podobno gibanje grafov pri datotekah, zopet nekoliko višje v sredini tedna. Predpostavljamo pa, da bodo časi bistveno hitrejši kakor pri oddajanju na strežnik.
- **Robni pogoji:** Na enak način smo tudi meritev pri prejemanju opravljali za vsako izmed obravnavanih velikosti datotek (1B, 10B, 100B, 1KB, 10KB, 100KB, 1MB), katere smo tudi prikazali z različnimi grafi. Pri prejemanju vsake datoteke smo počakali, da se je prenos uspešno zaključil, preden smo nadaljevali z zahtevo za prejemanje datoteke naslednje velikosti. Za vsako izmed izbranih velikost datotek smo dnevno naredili petnajst različnih meritev, pri katerih smo izmerili čas trajanja. Čas prejemanja smo izmerili s pomočjo statističnega opisa, ki nam ga ponuja *curl*. Vsaka meritev je bila opravljena ob naključnih časih, nedvisno od prejšnje. Meritve smo naredili neodvisno z dveh različnih lokacij in sicer deset meritev v okolici, preostalih pet pa v centru Ljubljanje. Na koncu smo izračunali še povprečni skupni čas obeh lokacij.
- **Komentar:** V grafu na sliki 5.7 je razvidno, da se je prejemanje vršilo v skladu s pričakovanji, namreč tudi tu datoteka velikosti 1MB nekoliko prevladuje. Zopet pa opazimo tudi skoke v sredini tedna in v prehodu iz nedelje na ponedeljek.



Slika 5.7: Trajanje prenosa skozi teden pri prejemanju.

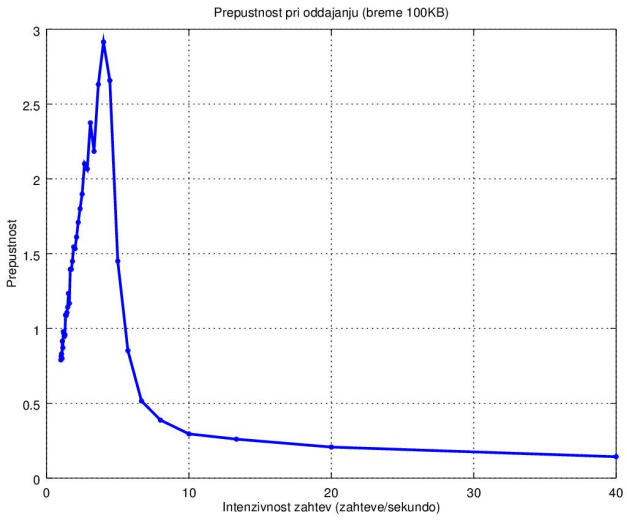
5.6.4 Iskanje točke nasičenja

Sledeča meritev predstavlja opazovanje prepustnosti obdelovanja zahtev ob spreminjanju intenzivnosti pošiljanja zahtevkov.

- **Hipoteza:** Naša predpostavka je, da bo sprva prepustnost naraščala sorazmerno z intenzivnostjo prihajanja zahtev, saj strežnik ne bi smel imeti težav z relativno nizkim številom zahtevkov. Pričakujemo pa, da se bo prepustnost ob večanju števila zahtevkov v časovni periodi sčasoma, na določeni točki uklonila navzdol.

- **Robni pogoji:** Za opazovanje odzivnosti in določitev točke nasičenja pri oddajanju smo si izbrali breme velikosti 100KB. Frekvenco oddajanja bremena smo skozi čas povečevali tako, da smo na vsakih trideset sekund zmanjšali zamik oddajanja za 0.025 sekunde. To smo počeli z namenom, da bi strežnik obremenili. Meritev pri vsaki različni odredbi frekvence oddajanja zahtevkov smo ponavljali v neprekinjenem časovnem odseku, v katerem smo kontinuitetno obremenjevali strežnik z velikim številom ponovitev posameznega zahtevka pri določeni intenzivnosti. S tem namenom smo skušali čim bolj eliminirati odklonske meritve ter se kar se da dobro približali povprečni vrednosti. Z orodjem *curl* smo izmerili čase posameznih prenosov, katere smo nato pri vsaki izmed intenzivnosti povprečili. Celotno meritev smo trikrat ponovili v različnih dnevih v tednu ob neodvisnih časih. Končni rezultat smo tako dobili s povprečenjem meritev.

- **Komentar:** Kot rezultat lahko tako na sliki 5.8 opazimo, kako se sprva prepustnost skoraj popolnoma proporcionalno povečuje z intenzivnostjo zahtev. Ob nekoliko večjem številu prihajanja zahtev pa lahko vidimo, da se približujemo točki, kjer se začne prepustnost ustaljevati. Ob večanju intenzivnosti zahtev pa vidimo, kako se prične prepustnost celo zmanjševati. Pri 4 zahtevah na sekundo zasledimo odgovore strežnika tipa `429 too_many_write_operations/`, kjer nam tudi predpiše čas premora za ponovitev zahteve. Pri 10 do 20 zahtevah na sekundo pa strežnik občasno celo prezira naše zahteve.



Slika 5.8: Prepustnost pri prenosu na strežnik.

Meritev predstavlja opazovanje prepustnosti obdelovanja zahtev ob spreminjanju intenzivnosti prejemanja zahtevkov.

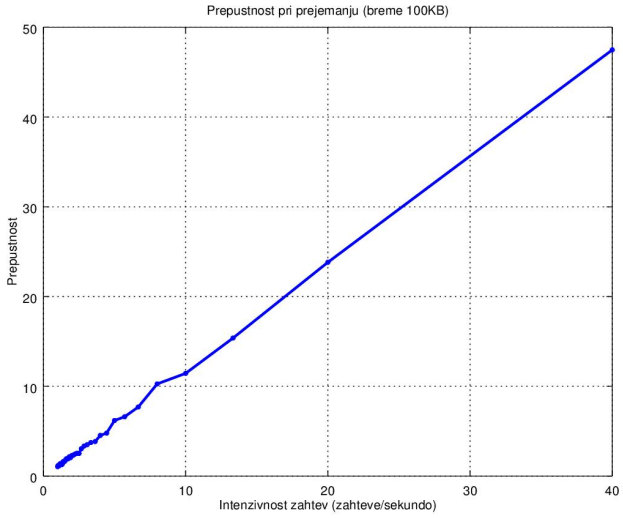
- **Hipoteza:** Podobno tudi tu pričakujemo, da bo prepustnost naraščala sorazmero z intenzivnostjo prihajanja zahtev. Pričakujemo pa, da bo točka, kjer se prepustnost ukloni, nastopila dosti kasneje. To predpostavljamo iz dejstva, da strežnik pri oddajanju ni toliko obremenjen, ker nam mora datoteko le posredovati.

- **Robni pogoji:** Pri opazovanju odzivnosti in določitvi točke nasičenja pri prejemanju smo si sprva prav tako izbrali breme velikosti 100KB, katerega smo povečali tudi na 1MB zaradi boljše opredelitve meritve. Z namenom, da bi strežnik obremenili, smo postopoma povečevali intenzivnost prihajanja zahtev. Frekvenco prejemanja bremena smo skozi čas povečevali tako, da smo na vsakih trideset sekund zmanjšali zamik prejemanja za 0.025 sekunde. S tem smo nameravali obrementiti strežnik. Tudi tu smo meritev pri vsaki različni odredbi intenzivnosti prejemanja zahtevkov ponavljali v neprekinjenem časovnem odseku, kjer smo neprestano obremenjevali strežnik z velikim številom ponovitev posameznega zahtevka pri določeni intenzivnosti. S tem želimo čim bolj zmanjšati odklonske meritve ter se kar se da dobro približali povprečni vrednosti. Z orodjem *curl* smo izmerili čase posameznih prenosov, katere smo nato pri vsaki izmed intenzivnosti povprečili. Celotno meritev smo prav tako trikrat ponovili v različnih dnevih v tednu, ter ob neodvisnih časih in smo za končni rezultat povprečili te dobljene meritve ob neodvisnih časih.

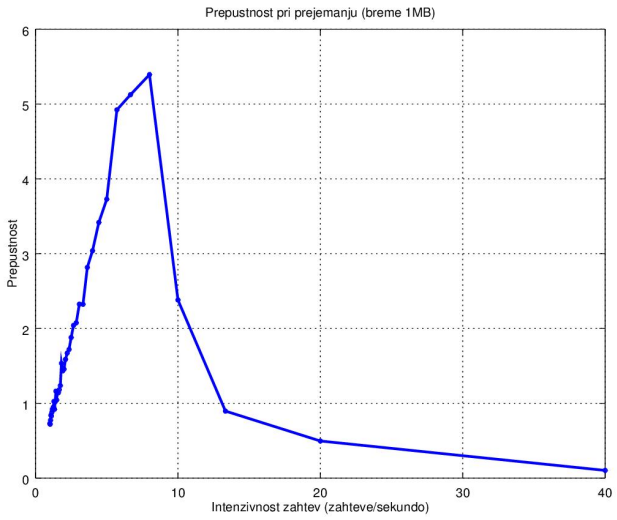
- **Komentar:** Podbno kot pri oddajanju, opazimo tudi na sliki 5.9 pri

prejemanju sprva sorazmerno rast prepustnosti ob večanju intenzivnosti zahtev, nakar pa se ob približevanju določene točke začne ustaljevati.

Še bolj podrobno vidimo to na sliki 5.10, kjer smo si za breme izbrali datoteko velikosti 1MB. S tem smo lahko nekoliko bolj obremenili strežnik pri prejemanju in defnirali uklon prepustnosti. Podobno kot pri oddajanju, tudi tu opazimo, kako z večanjem intenzivnosti zahtevkov prepustnost začne upadati. Pri 40 zahtevah na sekundo občasno zasledimo odgovore strežnika tipa `429 too_many_requests/`, s katerim nam tudi predpiše čas premora za ponovitev zahteve.



Slika 5.9: Prepustnost pri prenosu s strežnika.



Slika 5.10: Prepustnost pri prenosu s strežnika.

5.6.5 Prenos večjih datotek

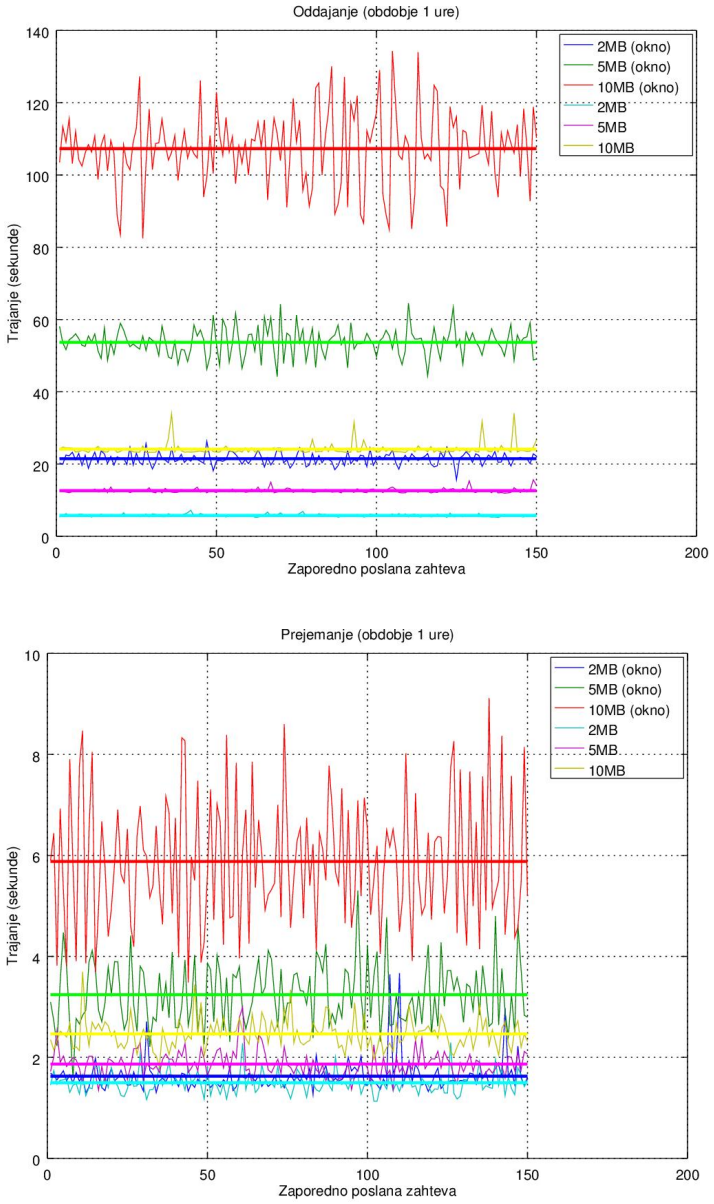
Ker pri splošni uporabi delamo tudi z datotekami, ki so večjih velikosti, smo se odločili primerjati obdelovanje zahtev večjih velikosti.

- **Hipoteza:** Podobno kot pri manjših velikostih pričakujemo, da bodo večje datoteke potrebovale več časa. Predpostavljamo tudi, da bodo datoteke, prenešene v načinu okna, potrebovale več časa.

- **Robni pogoji:** Osredotočali smo se na bremena velikosti 2MB, 5MB in 10MB. Prenose smo vršili na dva načina. V prvem načinu smo prenos izvedli tako, da smo ob vsakem prenosu datoteke počakali, da se je prenos uspešno zaključil, preden smo nadaljevali s prenosom datoteke naslednje velikosti. V drugem načinu pa smo prenos vršili z oknom (velikost 5). S tem smo konstantno imeli aktivnih ravno toliko povezav, kot je bilo okno veliko. Meritev za vsako izbrano velikost smo opravljali v neprekinjenem času ene ure. Z orodjem *curl* pa smo ob vsakem prenosu zabeležili čas trajanja. Celotno meritev smo opravili enkrat v enem dnevu tedna (torek). Najprej v prvem in nato en teden kasneje v drugem načinu.

- **Komentar:** Pri oddajanju na sliki 5.11 zgoraj vidimo, da je bila naša predpostavka pravilna. Opazimo pa, da se je breme velikosti 2MB v načinu z oknom (velikosti 5) v povprečju preneslo hitreje kot pa breme velikosti 10MB v navadnem načinu. Podobno opazimo tudi pri prejemanju na sliki 5.11 spodaj. Opazimo, da je prenos bremena velikosti 2MB v načinu z oknom (velikosti 5) hitrjše od navadnega načina pri bremenih velikosti

5MB in 10MB. Iz tega lahko sklepamo, da je Dropbox optimiziran za skupinsko delo nad datotekami takih velikosti.



Slika 5.11: Trajanje prenosa.

5.6.6 Uporabniška izkušnja ob obremenitvi

Da bi ugotovili še kakšna je uporabniška izkušnja med podobno obremenitvijo, smo se odločili simulirati čim bolj realno robno situacijo.

- **Hipoteza:** Predpostavljamo, da ob manjši obremenitvi motenj ne bomo čutili. Ob večji obremenitvi pa pričakujemo zmanjšano odzivnost osnovnih funkcij.
- **Robni pogoji:** Merjenja smo se lotili na tak način, da smo eksperiment razdelili na dva logična dela, katera smo ločili tudi lokacijsko. Na enem delu je potekala analiza povprečne uporabniške izkušnje v delu s storitvijo Dropbox. Povprečna uporabniška izkušnja zajema delo s prenosi raznih datotek (video, glasba, dokumenti in stisnjene datoteke) z osebnega računalnika na strežnik, prenosi s strežnika na osebni računalnik, uporabo predogleda naloženih datotek in morebitno sinhronizacijo. Tu skušamo ugotoviti kakovost uporabniške izkušnje, torej hitrost delovanja, odzivnost in vsesplošno izvajanje. Torej skušamo subjektivno ugotoviti kakšna bo naša izkušnja. Na drugem delu pa se izvaja obremenjevanje, s katerim skušamo vplivati na delovanje in posledično na uporabniško izkušnjo. Obremenjevanje je podobne oblike kot pri iskanju točke nasičenja, torej z bremenom datoteke velikosti 100kB, smo postopoma povečevali frekvenco oddajanja zahtev. Pri vsaki izbrani frekvenci oddajanja zahtev smo opazovali celotno opredeljeno uporabniško izkušnjo. Opazovali smo hitrost delovanja, odzivnost sistema na naše ukaze in ali so naši zadani ukazi sploh bili izvršeni. Sprva smo naredili kontrolni eksperiment, ki nam je dal grobo idejo o nevtralnem delovanju. Pri tem eksperimentu strežnika nismo obremenjevali iz večih lokacij, ampak zgolj izstavljali ukaze iz enega naslova.
- **Komentar:** Ugotovili smo, da neglede na frekvenco oddajanja zahtev večina lastnosti ni bila spremenjena. Torej ni bilo opažene razlike v sinronizaciji datotek, ali njihovih predogledih na strežniku, prav tako pa se ni bistveno spremenila hitrost prenosa datotek s strežnika. Kar pa je bilo spremenjeno, so hitrosti prenosov datotek na strežnik in sicer bolj kot smo povečevali frekvenco zahtev, bolj je hitrost prenosa na strežnik slabela. Ob takem delovanju je postala slaba tudi odzivnost strežnika na različne ukaze, hkrati pa se nekateri ukazi občasno sploh niso zaznali in izvedli. Pri 5 zahtevkih oddajanja omenjenega bremena na sekundo, smo imeli še sprejemljivo uporabniško izkušnjo, medtem ko smo pri 10 naleteli tudi na zavrnitve prenosov na strežnik, kar nam seveda uporabniško izkušnjo kvari. S tem smo dosegli in razbrali grobe ocene za robne obremenitve strežnika, kjer je uporabniška izkušnja še sprejemljiva.

5.7 Zaključek

Dropbox uporabljava v prostem času že dlje časa, saj nudi dobro sinhronizacijo datotek preko naprav, hitre prenose in pa zanesljivo delovanje. Zato sva se odločila, da ga pri tem predmetu tudi testirava. Dropbox je uporabniku prijazna oblačna storitev, z velikim številom podprtih fukncionalnosti, ki so uporabniku na voljo za oblačno shranjevanje datotek. Uporabnikov repozitorij Dropbox je dostopen iz večih naprav, ki lahko bazirajo na iOS, Android, blackberry, MacOS, Linux ali Windows.

Uporabljala sva prosto dostopno različico oblačne storitve Dropbox. Za uporabo brezplačne oblačne storitve Dropbox potrebujemo dostop do interneta, in pa uporabniški račun, ki ga je moč narediti kar na njihovi strani, s pomočjo dveh enostavnih korakov. Izvajala sva različne teste prenosa datotek, ob različnih časih, da sva razbrala odzivnost strežnika glede na čas izdane poizvedbe, teste prenosa različnih velikosti datotek, kjer sva očitala hitrosti prenosa datotek, teste prenosa datotek iz različnih fizično ločenih lokacij, kjer sva razbrala dostopnost oblačne storitve in pa teste, kjer sva vključila vse prej omenjene dimenzije.

Pri testiranju oblačne storitve Dropbox sva se naučila veliko. Kako se uporablja Dropboxov aplikacijski vmesnik, kako klient zaprosi strežnik za poizvedbo, kako klient izve informacije o njegovem datotečnem sistemu in podobno. Izvedeni testi so bili specificirani tako, da prikažejo čim bolj realno sliko vsakdanjega delovanja oblačne storitve Dropbox. Za primerjavo najinih rezultatov, bi navedla več različnih virov, kot so testiranje hitrosti prenosa [51], primerjalna študija oblačnih storitev [52], itd. Priporočava pa tudi gradivo drugih raziskovalcev omenjeno na [53] in [54], katerega sva tudi sama pregledala.

Dropbox se izkaže za več kot dobro oblačno storitev, saj nudi uporabnikom hitre in zanesljive funkcionalnosti prenosa datotek in sinhronizacije. Oblačna storitev Dropbox je odlična in priznana rešitev namenjena vsakdanjim uporabnikom, za prenos datotek med napravami ali med skupinami ljudi.

Poglavje 6

Analiza zmogljivosti nekaterih oblačnih storitev

Karmen Gostiša, Žiga Korelc, Blaž Marolt, Tomas Štefe

6.1 Opis problema

V zadnjih letih se je z razvojem kamer na pametnih telefonih zgodil izjemen porast multimedijskih vsebin. Tako danes že vsak posameznik mimogrede posname fotografijo ali video in ga nato shrani v oblačno hrambo podatkov.

Pojavila se je tudi težnja, da bi svoje podatke zavarovali na vse mogoče načine, od redundantnih polj trdih diskov do magnetnih trakov. V tem primeru predstavlja oblačna hramba podatkov eno izmed možnih rešitev.

V našem poglavju se bomo osredotočili na nekatere oblačne načine shrambe podatkov in jih primerjali med seboj. Namen tega poglavja je analizirati zmogljivost določenih ponudnikov oblačnih storitev in izbrati najboljšega med njimi glede na izbrano metriko. Pri izbiri najboljšega bomo prav tako upoštevali, kaj nam je ponujeno pri brezplačni obliki ponudnikove storitve - shranjevanju podatkov v oblak.

6.2 Izbira ponudnikov

Na podlagi naših izkušenj z uporabo oblačnih storitev smo izbrali Google Drive, Mego in Dropbox.

Google Drive

Google Drive je storitev za shranjevanje in sinhronizacijo datotek, ki jo je razvilo podjetje Google leta 2012. Storitev uporabnikom omogoča shranjevanje datotek v oblaku, njihovo sinhronizacijo med številnimi napravami in deljenje datotek z drugimi uporabniki [55].

Poleg spletne verzije Google Drive ponuja aplikacijo, ki omogoča dostop do datotek brez internetnega dostopa za računalnike z operacijskim sistemom Windows ali Mac in za pametne telefone z operacijskim sistemom Android in iOS. Google Drive zajema Google Docs, Sheets in Slides in pisarniške programe, ki omogočajo skupinsko urejanje dokumentov vseh vrst.

V brezplačnem načinu nam Google Drive ponuja 15 GB prostora v oblaku. Če želimo več oblačnega prostora, je na voljo plačljiva ponudba petih velikostnih razredov in sicer 100 GB, 1 TB, 10 TB, 20 TB in 30 TB. Največja velikost naložene datoteke je 5 TB [56].

Google ima veliko podatkovnih centrov po Evropi, Aziji in Ameriki [57]. S pomočjo Wiresharka smo ugotovili, da so bili naši podatki pri testiranju poslani na Mountain View v Kalifornijo, kar je razumljivo glede na to, da smo pri testiranju uporabljali brezplačen način shrambe podatkov.

Mega

Mega je oblačna storitev za hranjenje podatkov. Ustanovil jo je kontroverzni Kim Dotcom in je na voljo na platformah Windows Phone, Windows, Android, Blackberry 10 in iOS [58].

Mega je eden izmed prvih ponudnikov, ki je datoteke začel kriptirati tudi lokalno, preden jih prenesemo na strežnik [58]. Svojo storitev nudi brezplačno in proti plačilu. V brezplačnem načinu smo omejeni s 50 GB, pri plačljivih storitvah pa imamo na voljo velikostne razrede 200 GB, 500 GB, 2 TB in 4 TB [59].

Avgusta 2016 je imela Mega registriranih 50 milijonov uporabnikov iz več kot 245 držav in na strežnike prenesenih že več kot 20 milijard datotek [58].

Lokacije strežnikov pri Megi ne objavljajo javno. Pri našem testiranju smo po navedbah Wiresharka podatke pošiljali v Luksemburg.

Dropbox

Dropbox je brezplačna storitev, ki nam omogoča shranjevanje in dostop do dokumentov, fotografij, video posnetkov in drugih datotek. V svojem oblačnem prostoru lahko izdelamo poljubno število map, ki jih lahko tudi delimo z drugimi. Do njegovih storitev lahko dostopamo preko platform Windows, Mac, Linux, iOS, Android, Windows Phone ali preko spletnih brskalnikov.

Dropbox ima ogromno uporabnikov. Ustanovljen je bil leta 2007 in že aprila 2010 presegel mejnik enega milijona uporabnikov. Marca 2016 jih je imel že več kot 500 milijonov [60].

Dropbox uporablja poslovni model freemium. Pri najbolj osnovnem brezplačnem paketu lahko uporabnik upravlja z 2 GB oblačnega prostora, ki ga

lahko razširi za vsakega povabljenega prijatelja. Posameznik lahko z mesečno naročnino uporablja tudi Dropbox Plus. Za 8.25 EUR/mesec uporabnik pridobi 1 TB prostora in še nekaj dodatnih funkcionalnosti. Ponuja tudi možnost oddaje oblačnega prostora za skupine. Loči tri plačilne skupine: Standard (10 EUR/uporabnik/mesec), Advanced (15 EUR/uporabnik/mesec) in Enterprise (cena po dogovoru). Standard nudi 2 TB prostora, ostala dva pa "neskončno" veliko prostora [61].

Dropbox ima nekaj lastnih strežnikov postavljenih po Ameriki, v Evropi pa uporablja Amazon Web Services [62], ki pa ima strežnike postavljene na Irskem, v Frankfurtu in v Londonu [63]. Wireshark nam je pokazal, da smo pri testiranju podatke pošiljali v Frankfurt.

6.3 Izbira tehnologij

V tem razdelku so opisane tehnologije, s katerimi smo merili zmogljivost izbranih oblačnih storitev.

C#

C# je splošnonamenski programski jezik, ki obsega deklarativno, funkcijsko in objektno orientirano programiranje. Microsoftovi razvijalci so se zgledovali po jezikih C, C++ in Javi ter od njih uporabili le najboljše značilnosti. C# cilja na izvajalsko okolje CLI in s tem ogrodje .NET [64].

Google Drive API

Google Drive API je del sklopa Googlovih API-jev, ki omogočajo komunikacijo z Googlovimi storitvami in njihovo integracijo v druge storitve [65]. Ključni funkcionalnosti sta nalaganje datotek na Google Drive in prenos iz njega. Poleg tega ponuja še druge funkcije:

- iskanje datotek - ob prenosu datoteke na strežnik Google Drive avtomatsko indeksira datoteke pogostih formatov: .html, .xml, .txt itd.;
- deljenje in sodelovanje - Google nam omogoča, da v naši aplikaciji prikažemo standardni Google Drive Share dialog, v katerem uporabniki enostavno delijo datoteke.;
- uporaba bližnjic - naša aplikacija lahko na Google Drive ustvari bližnjico do datotek, ki so shranjene izven oblaka, vendar jih lahko še vedno odpiramo, indeksiramo, iščemo in delimo kar na Google Drive.;

Mega API

Mega API je veliko manj zmogljiv, a vseeno ponuja dovolj osnovnih funkcionalnosti za delo z oblačno storitvijo Mega. V našo aplikacijo lahko tako vključimo avtentikacijo uporabnika, spremljanje porabe prostora, prenašanje datotek na

in iz oblaka. Vse funkcionalnosti so opisane skupaj s primeri na strani Github [66].

Dropbox API

Najpomembnejši funkcionalnosti Dropbox API-ja sta nalaganje na Dropbox in prenos iz njega. Poleg tega omogoča še naslednje funkcionalnosti [67], [68]:

- pridobivanje metapodatkov posameznih datotek,
- deljenje map/datotek,
- pridobivanje url povezave deljenih map/datotek,
- pridobivanje uporabniških podatkov,
- izpis shranjenih datotek/map na uporabnikovem oblačnem prostoru,
- brisanje/posodabljanje datotek/map.

Wireshark

Wireshark je brezplačni program za analiziranje paketov, ki potujejo po omrežju. Veliko se ga uporablja pri analizi omrežja, odpravljanju napak in razvijanju komunikacijskih aplikacij [69].

6.4 Opis sistema za testiranje

Napisali smo program v jeziku C#, ki teče na strani odjemalca in za svoje delovanje uporablja API-je, ki poskrbijo za vzpostavitev komunikacije z oblačno storitvijo. Odjemalec v različnih testnih scenarijih pošilja bremena, program pa izmeri čas odgovora sistema (glej sliko 6.1).

Nalaganje datotek na strežnik

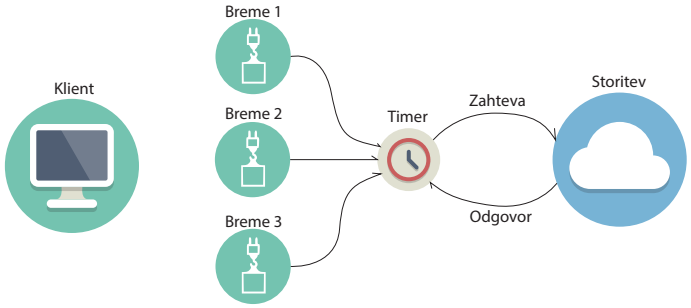
Z zahtevkom HTTP POST odjemalec pošlje datoteko na strežnik, ki datoteki dodeli identifikator in ga nato vrne v odgovoru na zahtevek. Ta identifikator je potreben, če odjemalec kasneje želi datoteko prenesti s strežnika.

Prenos datotek iz strežnika

Odjemalec pošlje zahtevek HTTP GET z identifikatorjem datoteke. Strežnik poišče datoteko in jo odpošlje odjemalcu. V primeru, da datoteke ne najde, odjemalec prejme sporočilo o napaki.

Metrika

Pri vseh oblačnih ponudnikih smo merili čas odgovora. To je čas, ki je potreben za to, da uporabnik datoteko naloži ali jo prenese. Zajema čas pošiljanja zahtevka, čas obdelave zahtevka in čas pošiljanja odgovora.



Slika 6.1: Sistem za testiranje oblačnih storitev.

6.5 Eksperimenti

Za izbor najboljšega izmed oblačnih ponudnikov smo zastavili več različnih eksperimentov. Pri njih nas je zanimal čas, ki ga potrebujejo strežniki za odgovor in čas nalaganja ter prenašanja v odvisnosti od velikosti in števila datotek. Za bolj točne rezultate smo vse teste znotraj eksperimenta ponovili trikrat in meritve povprečili. Vsa testiranja so potekala na povezavi 50/20 Mbit/s.

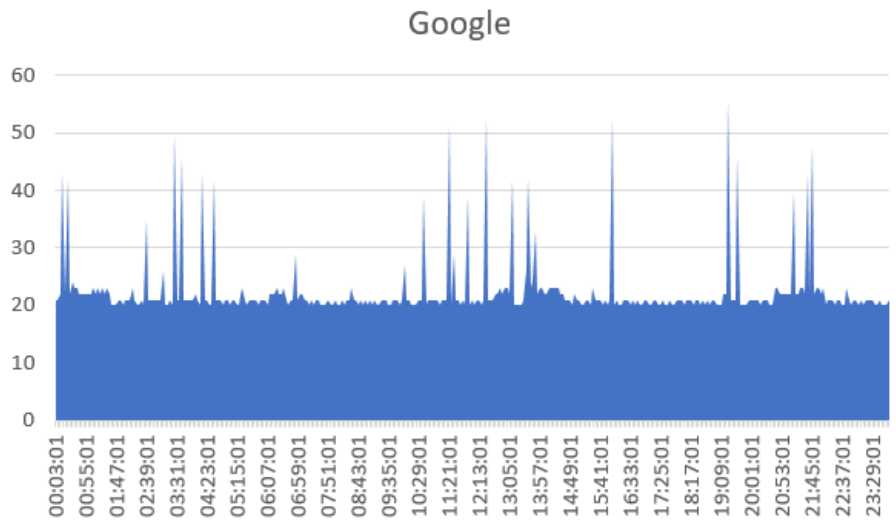
Naša splošna hipoteza, ki velja za vse eksperimente je, da bo na čas odgovora najbolj vplivala lokacija postavljenih strežnikov in sicer, bližje kot je strežnik, krajši bo čas odgovora. Po tej teoriji bi moral najnižje čase imeti Dropbox s strežniki v Frankfurtu, nato Mega s strežniki v Luksemburgu in na koncu Google Drive s strežniki v Kaliforniji.

6.5.1 Dostopnost

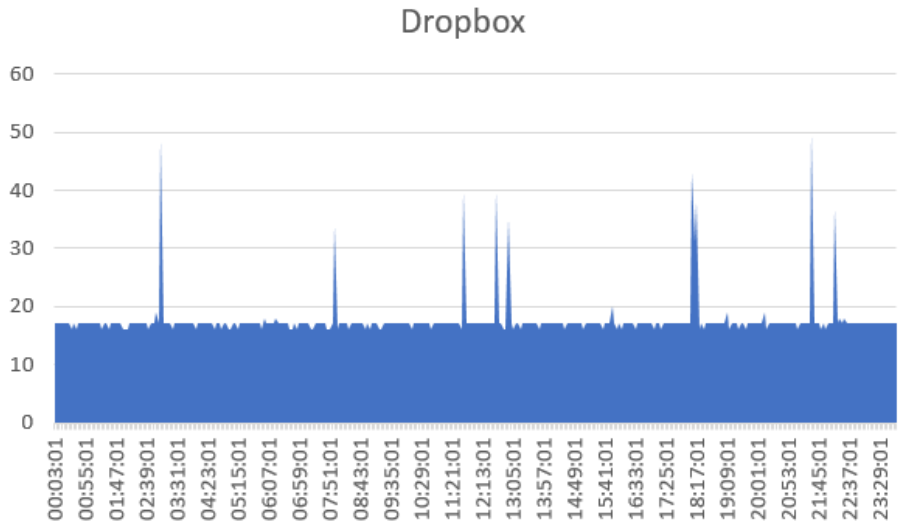
Za vsakega oblačnega ponudnika smo določili časovni interval, ko sta strežnik in omrežje najmanj obremenjena. To smo storili z namenom, da bomo ob tisti uri izvedli nadaljnje eksperimente, ki zajemajo nalaganje datotek na strežnik in prenašanje iz njega ter tako dobili najboljše rezultate.

Test smo izvršili tako, da smo s pomočjo skripte bash 24 ur vsako minuto na strežnike poslali zahtevek ping in čase beležili. To smo počeli 22. 5. 2017, 23. 5. 2017 in 24. 5. 2017. Na slikah 6.2, 6.3 in 6.4 so prikazani povprečni časi (v milisekundah) teh treh dni.

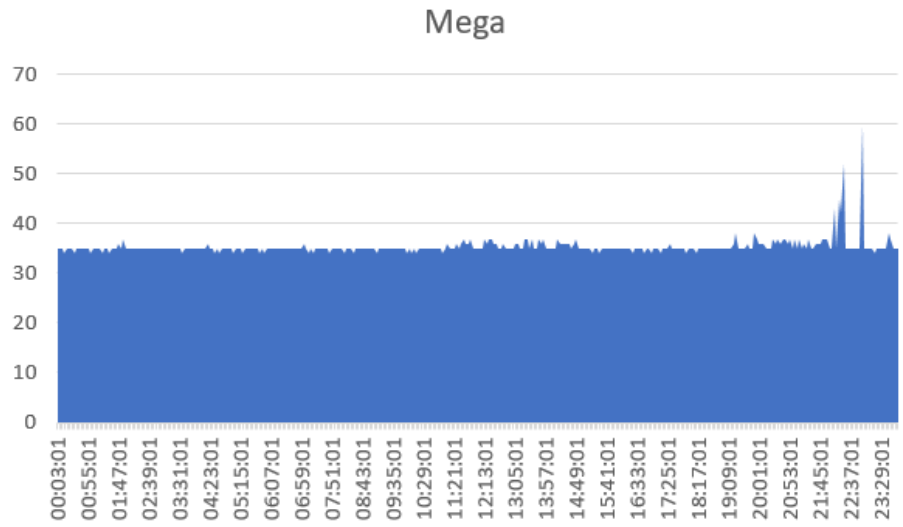
Rezultati



Slika 6.2: Povprečni čas odgovora ponudnika Google Drive (Kalifornija).



Slika 6.3: Povprečni čas odgovora ponudnika Dropbox (Frankfurt).



Slika 6.4: Povprečni čas odgovora ponudnika Mega (Luksemburg).

Pri ponudnikih Google Drive in Dropbox večjih odstopanj med posameznimi deli dneva ni (glej sliki 6.2 in 6.3). Pri obeh sicer lahko opazimo manjšo podobnost in sicer povečano zasedenost strežnikov med sledečimi časovnimi intervali:

- 2:00 in 3:00,
- 6:30 in 7:30,
- 11:00 in 13:00,
- 17:30 in 18:40,
- 21:00 in 22:00.

Vseeno so rezultati tako nihajoči, da boljših zaključkov o največji dnevni obremenjenosti strežnikov teh dveh ponudnikov ne moremo narediti. Do takšnih rezultatov je najverjetneje prišlo, ker sta ponudnika svetovno priznana in med najbolj uporabljanimi ponudniki oblačnih storitev [70]. Meritve smo vseeno na podlagi tega testa za ta dva ponudnika izvedli med 16. in 17. uro.

Po rezultatih, ki nam jih je vrnil Google Drive (glej tabelo 6.2, lahko sklepamo, da v resnici zahtevkov ping nismo pošiljali v Kalifornijo, saj bi v tem primeru časi morali biti večji (okoli 35 ms). Razlog za manjše čase je v tem, da se vse TCP povezave zaključijo na najbližjem Googlovem vozlišču, naprej pa je promet usmerjen preko Googlovega zasebnega omrežja [71].

Pri ponudniku Mega smo dobili bolj značilen graf, iz katerega je jasno razvidno, da so njihovi strežniki najbolj obremenjeni med 22. in 23. uro (glej sliko 6.4). Menimo, da največ uporabnikov Mege prihaja iz geografsko podobnega območja, ki storitev uporabljajo ob približno enaki uri, najverjetneje pozno

popoldne in zvečer. Ta podatek nam je koristil in za Mego smo zato vse eksperimente izvedli med 17. in 18. uro, ko so njihovi strežniki manj obremenjeni.

6.5.2 Nalaganje in prenašanje različno velikih datotek

Opis

Pri tem testu nas je zanimal čas odgovora v odvisnosti od velikosti datotek, ki jih je odjemalec pošiljal oz. prenašal s strežnika. Odjemalec je zaporedno poslal oz. prenesel k sebi na disk 10 datotek za vsako izbrano velikost (1 KB, 1 MB, 3 MB, 5 MB, 10 MB, 20 MB). Takšne velikosti v MB smo izbrali, ker smo želeli pridobiti rezultate za datoteke, ki so po velikosti najbolj podobne fotografijam in videom.

Pri nalaganju datotek nas je dodatno zanimal vpliv koščkov (ang. *chunks*) na čas nalaganja. Programski vmesniki za delo z izbranimi oblačnimi storitvami pred prenosom na strežnik datoteke "razkosajo"na manjše dele, da zagotovijo bolj optimalno pošiljanje. V primeru napake pri pošiljanju koščka jim namreč ni potrebno ponovno poslati cele datoteke, ampak samo tisti košček.

Programska vmesnika za Google Drive in Mego ponujata možnost nastavitve velikosti koščkov. Dropbox žal ne ponuja te možnosti, le v primeru nalaganj večjih od 150 MB svetuje uporabo metode UploadSessionStartAsync. Testirali smo torej le prva dva ponudnika in sicer smo za vsakega nastavili velikost koščkov na polovico manjšo y ter polovico večjo velikost z od privzete velikosti tega ponudnika. Privzeta velikost koščkov x je pri Google Drive 10 MB [72], Megi 1 MB in Dropboxu 4 MB [73] (glej tabelo 6.1).

velikost koščkov	x	y	z
Google Drive	10 MB	5 MB	15 MB
Mega	1 MB	0.5 MB	1.5 MB

Tabela 6.1: Privzeta velikost koščkov x, polovična privzeta velikost y in za polovico povečana privzeta velikost z.

Test smo izvedli dne 10. 5. 2017 ob 16. uri za ponudnika Google Drive in Dropbox ter ob 17. uri za Mego.

Hipoteze

Predpostavljamo, da se bo čas nalaganja in prenašanja datotek povečeval premo sorazmerno glede na velikost datoteke in da bodo časi nalaganja datotek večji kot časi prenašanja, saj naša internetna povezava znaša 50/20 Mbit/s.

Pričakujemo tudi, da bodo rezultati s privzeto velikostjo koščkov boljši kot z nastavljeno, saj so algoritmi, ki jih uporabljajo uporabniški programski vmesniki ponudnikov verjetno prilagojeni privzeti velikosti in za to velikost delujejo najhitreje.

Rezultati s privzeto velikostjo koščkov x

velikost datoteke	1 KB	1 MB	3 MB	5 MB	10 MB	20 MB
nalaganje:						
čas 1	11.87 s	13.97 s	23.32 s	31.38 s	58.52 s	102,29 s
čas 2	11.84 s	13.44 s	23.89 s	32.96 s	62.16 s	107,87 s
čas 3	11.88 s	13.53 s	23.32 s	34.44 s	63.06 s	103,21 s
povprečni čas	11.86 s	13.64 s	23.51 s	32.93 s	61.25 s	104,46 s
prenašanje:						
čas 1	4.99 s	9.48 s	18.36 s	26.13 s	47.73 s	104.96 s
čas 2	5.04 s	9.37 s	17.76 s	26.18 s	41.18 s	114.65 s
čas 3	5.54 s	9.37 s	16.30 s	26.45 s	41.12 s	108.75 s
povprečni čas	5.19 s	9.41 s	17.48 s	26.26 s	43.34 s	109.45 s

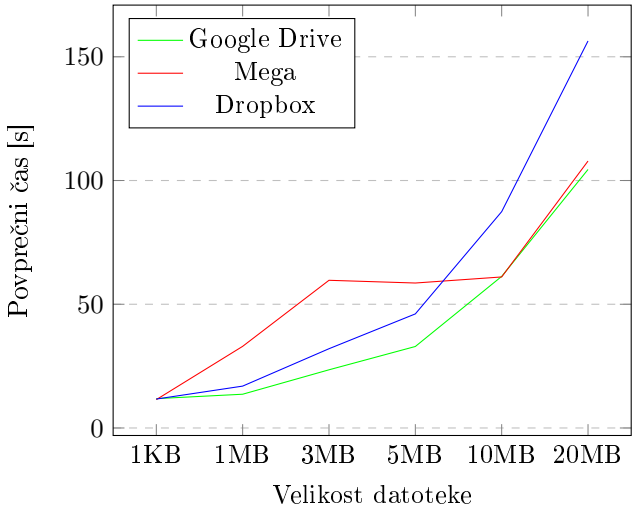
Tabela 6.2: Rezultati ponudnika Google Drive (Kalifornija, x = 10 MB).

velikost datoteke	1 KB	1 MB	3 MB	5 MB	10 MB	20 MB
nalaganje:						
čas 1	12.65 s	34.67 s	39.26 s	80.39 s	57.48 s	112.88 s
čas 2	9.82 s	36.25 s	72.55 s	31.42 s	69.50 s	104.92 s
čas 3	11.90 s	28.04 s	67.23 s	63.93 s	56.10 s	105.82 s
povprečni čas	11.46 s	32.99 s	59.68 s	58.58 s	61.03 s	107.87 s
prenašanje:						
čas 1	16.04 s	24.03 s	30.75 s	35.69 s	65.60 s	98.96 s
čas 2	13.17 s	23.23 s	40.16 s	30.37 s	53.07s	130.41 s
čas 3	15.90 s	25.65 s	28.93 s	53.89 s	58.09 s	112.82 s
povprečni čas	15.04 s	24.30 s	33.28 s	39.99 s	58.92 s	114.06 s

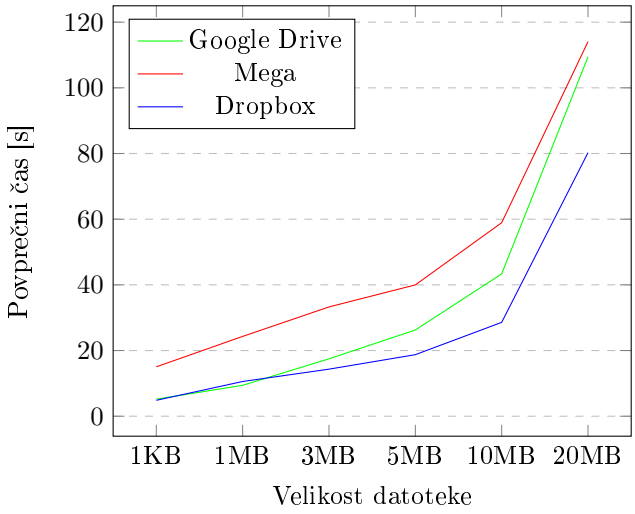
Tabela 6.3: Rezultati ponudnika Mega (Luksemburg, x = 1 MB).

velikost datoteke	1 KB	1 MB	3 MB	5 MB	10 MB	20 MB
nalaganje:						
čas 1	12.06 s	17.42 s	34.98 s	45.99 s	88.86 s	162.59 s
čas 2	9.82 s	13.66 s	28.04 s	45.75 s	85.07 s	150.85 s
čas 3	11.16 s	19.68 s	33.09 s	46.55 s	88.34 s	155.73 s
povprečni čas	11.69 s	16.92 s	32.03 s	46.10 s	87.42 s	156.39 s
prenašanje:						
čas 1	5.76 s	10.52 s	16.39 s	18.34 s	27.09 s	79.23 s
čas 2	4.48 s	11.47 s	10.82 s	18.68 s	30.07 s	80.35 s
čas 3	4.30 s	9.72 s	15.77 s	19.20 s	27.72 s	81.21 s
povprečni čas	4.84 s	10.57 s	14.33 s	18.74 s	28.58 s	80.26 s

Tabela 6.4: Rezultati ponudnika Dropbox (Frankfurt, x = 4 MB).



Slika 6.5: Graf povprečnega časa nalaganja datotek s privzeto velikostjo koščkov x v odvisnosti od velikosti datoteke.



Slika 6.6: Graf povprečnega časa prenašanja datotek s privzeto velikostjo koščkov x v odvisnosti od velikosti datoteke.

Iz rezultatov je razvidno, da čas nalaganja datotek narašča skoraj linearno (glej graf 6.5). Pri datotekah velikih 1 KB so časi vseh ponudnikov podobni (glej tabele 6.2, 6.4 in 6.3). Že pri 1 MB pa pride do večjih razlik, ki se nato le stopnjujejo. Zanimivo je, da ima Mega pri nalaganju datotek velikih 10 MB najboljši čas med vsemi ponudniki. Ta čas je tudi zelo podoben Meginem času nalaganja datotek velikih 3 MB in 5 MB. Takšen rezultat bi si lahko dobro

razlagali, če bi bila Megina privzeta velikost koščkov 10 MB.

Dropbox ima pri vseh velikostih datotek najmanjši čas prenašanja (glej tabelo 6.4). Do datotek velikosti 3 MB je zelo podoben ponudniku Google Drive (glej tabelo 6.2), pri večjih pa se odreže za dobro tretjino bolje kot Google Drive.

Potrdimo lahko naši hipotezi, saj se vsem ponudnikom čas nalaganja in prenašanja povečuje približno premo sorazmerno z velikostjo datotek in vsi časi nalaganj so v splošnem večji kot pri prenašanju datotek.

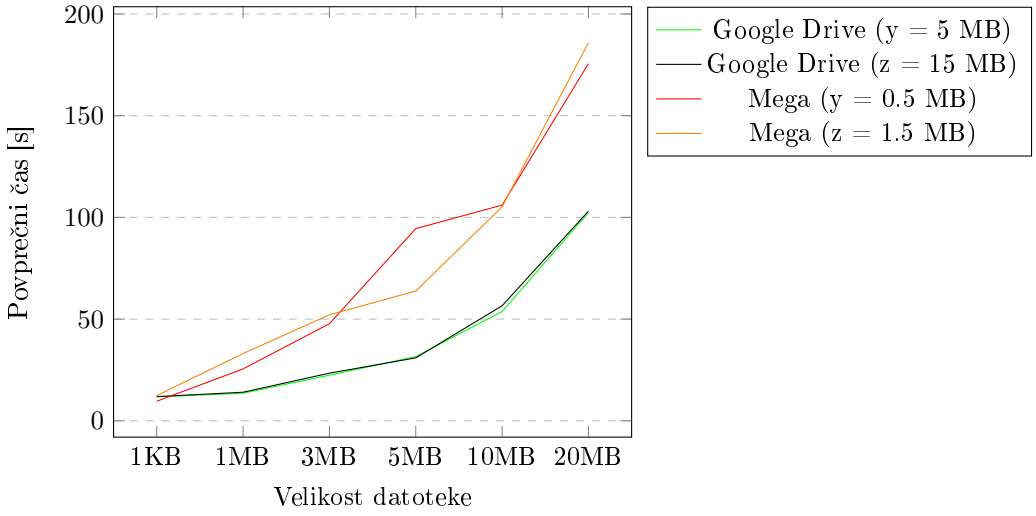
Rezultati nalaganja datotek z nastavljeno velikostjo koščkov na y in z

velikost datoteke	1 KB	1 MB	3 MB	5 MB	10 MB	20 MB
y = 5 MB						
čas 1	11.76 s	13.03 s	22.54 s	32.24 s	54.23 s	100.56 s
čas 2	11.99 s	13.87 s	22.76 s	32.22 s	52.87 s	101.83 s
čas 3	11.57 s	13.88 s	22.03 s	30.24 s	54.11 s	104.51 s
povprečni čas	11.77 s	13.57 s	22.44 s	31.57 s	53.73 s	102.30 s
z = 15 MB						
čas 1	12.27 s	14.16 s	22.80 s	31.31 s	56.04 s	102.47 s
čas 2	11.91 s	14.24 s	23.85 s	30.89 s	56.88 s	102.23 s
čas 3	11.47 s	13.76 s	23.59 s	30.59 s	56.77 s	104.51 s
povprečni čas	11.89 s	14.05 s	23.42 s	30.93 s	56.56 s	103.07 s

Tabela 6.5: Rezultati ponudnika Google Drive (Kalifornija).

velikost datoteke	1 KB	1 MB	3 MB	5 MB	10 MB	20 MB
y = 0.5 MB						
čas 1	8.54 s	40.00 s	39.16 s	64.07 s	113.95 s	169.77 s
čas 2	10.83 s	18.76 s	34.61 s	77.90 s	103.16 s	190.09 s
čas 3	9.46 s	17.76 s	69.38 s	141.47 s	101.01 s	166.89 s
povprečni čas	9.61 s	25.51 s	47.72 s	94.48 s	106.04 s	175.59s
z = 1.5 MB						
čas 1	9.48 s	41.92 s	51.38 s	76.84 s	103.29 s	182.91 s
čas 2	10.47 s	38.85 s	58.77 s	65.07 s	108.19 s	178.39 s
čas 3	17.25 s	18.42 s	46.25 s	49.46 s	104.14 s	196.47 s
povprečni čas	12.39 s	33.06 s	52.13 s	63.79 s	105.21 s	185.93 s

Tabela 6.6: Rezultati ponudnika Mega (Luksemburg).



Slika 6.7: Graf povprečnega časa nalaganja datotek z velikostjo koščkov y in z v odvisnosti od velikosti datoteke.

Presenetljivo se oba ponudnika v splošnem odrežeta bolje kot pri privzeti vrednosti koščkov (za primerjavo glej sliki 6.5 in 6.7). Google Drive ima rezultat slabši le pri datotekah velikosti 3 MB, Mega pa pri datotekah večjih od 10 MB.

Iz slike 6.7 razberemo, da ima Google Drive podobne čase pri obeh velikostih.

6.5.3 Nalaganje in prenašanje različnega števila datotek

Opis

Pri tem eksperimentu smo opazovali čas nalaganja in prenašanja v odvisnosti od števila poslanih datotek. Odjemalec je zaporedno poslal oz. si prenesel 5, 10, 20 in 50 datotek z velikostjo 1 MB. Pri nalaganju datotek smo pustili privzeto velikost koščkov.

Test smo izvedli dne 10. 5. 2017 ob 16. uri za ponudnika Google Drive in Dropbox ter ob 17. uri za Mego na povezavi 50/20 Mbit/s.

Hipotezi

Predvidevamo, da manj datotek kot je, hitreje bomo prejemali odgovore s strani sistema. Pričakujemo tudi, da bo čas prenašanja datotek manjši kot pri nalaganju.

Rezultati s privzeto velikostjo koščkov

št. datotek	5	10	20	50
nalaganje:				
čas 1	8.84 s	14.77 s	32.68 s	71.47 s
čas 2	11.30 s	13.90 s	27.90 s	70.21 s
čas 3	9.01 s	15.60 s	26.99 s	83.40 s
povprečni čas	9.72 s	14.76 s	29.19 s	75.03 s
prenašanje:				
čas 1	4.70 s	8.67 s	9.56 s	8.98 s
čas 2	5.22 s	8.92 s	9.32 s	9.18 s
čas 3	4.66 s	8.93 s	9.03 s	8.55 s
povprečni čas	4.86 s	8.84 s	9.30 s	8.91 s

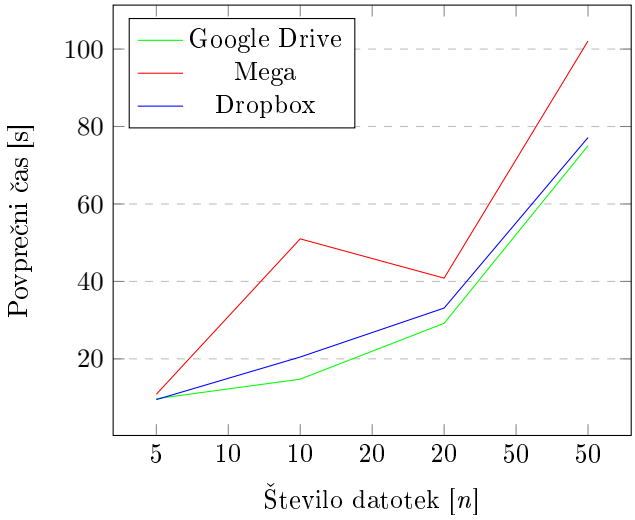
Tabela 6.7: Rezultati ponudnika Google Drive (Kalifornija, x = 10 MB).

št. datotek	5	10	20	50
nalaganje:				
čas 1	8.36 s	46.63 s	38.86 s	90.60 s
čas 2	12.25 s	12.62 s	36.24 s	94.93 s
čas 3	11.97 s	93.72 s	47.44 s	120.73 s
povprečni čas	10.86 s	50.99 s	40.85 s	102.09 s
prenašanje:				
čas 1	8.63 s	115.00 s	46.91 s	90.75 s
čas 2	9.50 s	18.32 s	38.77 s	94.13 s
čas 3	9.73 s	19.58 s	55.42 s	113.14 s
povprečni čas	9.29 s	50.96 s	47.04 s	99.34 s

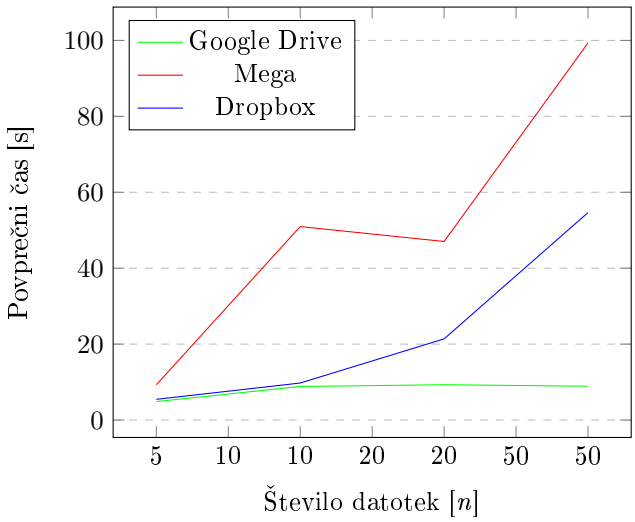
Tabela 6.8: Rezultati ponudnika Mega (Luksemburg, x = 1 MB).

št. datotek	5	10	20	50
nalaganje:				
čas 1	8.68 s	21.68 s	33.42 s	77.15 s
čas 2	9.00 s	18.57 s	33.53 s	77.19 s
čas 3	10.74 s	21.19 s	32.41 s	77.12 s
povprečni čas	9.48 s	20.48 s	33.12 s	77.15 s
prenašanje:				
čas 1	4.23 s	11.50 s	20.13 s	56.18 s
čas 2	6.93 s	7.35 s	20.17 s	53.22 s
čas 3	5.17 s	10.40 s	23.84 s	54.54 s
povprečni čas	5.45 s	9.75 s	21.38 s	54.65 s

Tabela 6.9: Rezultati ponudnika Dropbox (Frankfurt, x = 4 MB).



Slika 6.8: Graf povprečnega časa nalaganja datotek v odvisnosti od števila datotek.



Slika 6.9: Graf povprečnega časa prenašanja datotek v odvisnosti od števila datotek.

Čas nalaganja pri ponudnikih Google Drive in Dropbox je po naših pričakovanjih premosorazmeren s številom datotek. Nekoliko drugače pa je pri ponudniku Mega. Ta je za nalaganje in prenos 20 datotek porabil manj časa kot za 10 datotek enake velikosti (glej tabelo 6.8). V splošnem pa so časi znotraj ponovitve eksperimenta pri Megi zelo različni. Za nalaganje desetih datotek je namreč

enkrat porabila 12.62 s, drugič pa 93.72 s. Do takšnih anomalij je prihajalo, ker si Mega vzame pravico do zmanjšanja pasovne širine (ang. *bandwidth*) pri uporabnikih, ki uporabljajo brezplačno različico. To stori zato, da plačljivim uporabnikom zagotovi dobro povezavo [74].

Nekoliko nelogičen rezultat nam je ponudil tudi Google Drive (glej tabelo 6.7). Ta je za prenos 50 datotek potreboval 0.39 sekunde manj kot za prenos 20 datotek enake velikosti. Menimo, da je tudi tu prišlo do manjše anomalije zaradi hitrega zmanjšanja pasovne širine pri prenosu 20 datotek.

6.5.4 24-urno nalaganje datotek

Opis

Pri tem eksperimentu smo vsakemu oblačnemu ponudniku zaporedno pošiljali datoteko velikosti 1 MB in to počeli 24 ur. Velikost koščkov je privzeta.

S testom smo pričeli v soboto, 27. 05. 2017 ob 11. uri in zaključil naslednji dan ob isti uri. Testirali smo na povezavi 100/20 Mbit/s.

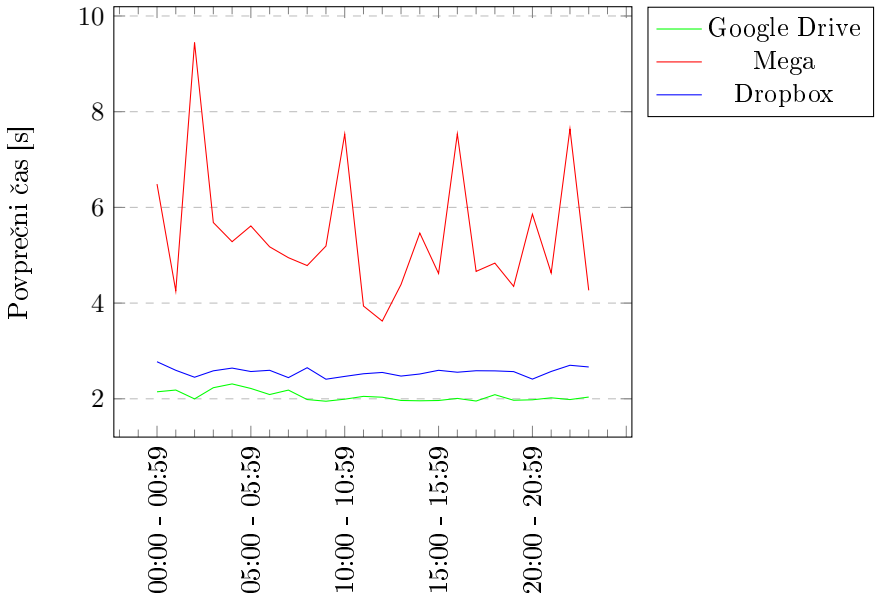
Hipoteza

Pričakujemo, da bodo iz grafa vidni manjši vzorci, kdaj je oblačna storitev najbolj zasedena in da bodo rezultati podobni rezultatom eksperimenta 6.5.1, kjer smo pošiljali zahtevke ping.

Rezultati

časovni interval	Google Drive	Mega	Dropbox
00:00 - 00:59	2.1461 s	6.4872 s	2.7745 s
01:00 - 01:59	2.1829 s	4.2539 s	2.5955 s
02:00 - 02:59	1.9978 s	9.4443 s	2.4507 s
03:00 - 03:59	2.2315 s	5.6835 s	2.5846 s
04:00 - 04:59	2.3108 s	5.2833 s	2.6417 s
05:00 - 05:59	2.2158 s	5.6124 s	2.5699 s
06:00 - 06:59	2.0890 s	5.1759 s	2.5955 s
07:00 - 07:59	2.1818 s	4.9489 s	2.4423 s
08:00 - 08:59	1.9844 s	4.7843 s	2.6494 s
09:00 - 09:59	1.9492 s	5.1919 s	2.4091 s
10:00 - 10:59	1.9917 s	7.5312 s	2.4679 s
11:00 - 11:59	2.0508 s	3.9385 s	2.5233 s
12:00 - 12:59	2.0329 s	3.6241 s	2.5507 s
13:00 - 13:59	1.9652 s	4.3887 s	2.4749 s
14:00 - 14:59	1.9581 s	5.4632 s	2.5181 s
15:00 - 15:59	1.9643 s	4.6190 s	2.5954 s
16:00 - 16:59	2.0079 s	7.5362 s	2.5552 s
17:00 - 17:59	1.9532 s	4.6624 s	2.5864 s
18:00 - 18:59	2.0860 s	4.8352 s	2.5835 s
19:00 - 19:59	1.9695 s	4.3501 s	2.5680 s
20:00 - 20:59	1.9790 s	5.8589 s	2.4114 s
21:00 - 21:59	2.0206 s	4.6296 s	2.5711 s
22:00 - 22:59	1.9841 s	7.6400 s	2.7006 s
23:00 - 23:59	2.0364 s	4.2668 s	2.6652 s

Tabela 6.10: Povprečni časi nalaganja datotek v danih časovnih intervalih.



Slika 6.10: Graf povprečnega časa nalaganja datotek v danih časovnih intervalih.

	Google Drive	Mega	Dropbox
skupni povprečni čas	2.05 s	5.27 s	2.56 s
največji čas	12.25 s	1686.74 s	7.66 s
najmanjši čas	1.30 s	1.75 s	1.36 s
99. percentil	2.69 s	25.78 s	4.23 s
99.9 percentil	3.23 s	306.05 s	5.39 s

Tabela 6.11: Skupni povprečni čas, največji čas in najmanjši čas nalaganja datotek ter 99. in 99.9 percentil.

Iz slike 6.10 je razvidno, da ima Mega zelo neenakomeren čas nalaganja datotek. Po podrobnem pregledu vseh meritev so izstopali nekateri veliki časi (60.04 s, 298.80 s, 1686.74 s). Tako kot smo ugotovili pri eksperimentu 6.5.3, tudi tu menimo, da je Mega takrat zmanjšala pasovno širino in plačljivim uporabnikom zagotovila boljšo povezavo [74].

Google Drive in Dropbox imata skozi dan precej enakomeren čas nalaganja datotek. Zanimivo je, da se od 2. ure do 4. ure zjutraj pri obeh ponudnikih časi nalaganja povečujejo, nato padajo do 7. ure zjutraj in nato ponovno povečajo. Rezultati so podobni eksperimentu 6.5.1, kjer smo ugotovili večjo zasedenost strežnikov ravno v teh časovnih intervalih. Sklepamo torej lahko, da so to intervali v dnevu, ko storitev uporablja največ ljudi.

Iz tabele 6.11 razberemo, da ima najmanjši skupni povprečni čas nalaganj datotek Google Drive, sledi Dropbox in nato Mega. Po naših pričakovanjih glede na prejšnje eksperimente, je največji čas nalaganja dosegla Mega s kar 1686.74 s (28.11 min). Tu je prišlo do večje anomalije. Razlog zanjo ni v zmanjšanju pasovne širine, saj je čas prevelik. Verjetno se je na strani odjemalca ali operaterja, na katerega je priključen odjemalec, močno zasičila povezava. Možno pa je tudi, da so bili ravno v tem trenutku Megini strežniki močno obremenjeni. Vendar nam je izračun 99. percentila pri Megi lahko v tolažbo, saj pove, da bomo le 1% časa čakali dlje kot 25.78 s. Pri Google Drive 99.9% časa ne bomo čakali dlje kot 3.23 s, pri Dropboxu pa 5.39 s.

6.6 Uporabniška izkušnja

Pri vrednotenju kakovosti oblačnega ponudnika je zelo pomembna uporabniška izkušnja. Stik uporabnika z oblačno storitvijo predstavlja njen grafični uporabniški vmesnik, zato je pomembno, da podpira odpiranje različnih formatov datotek in pri tem uporabniku ponudi prijetno izkušnjo.

Opis

Za oceno kakovosti uporabniške izkušnje smo v spletni različici izbranih oblačnih ponudnikov odprli naslednje datoteke:

- JPG z resolucijo 6000x4000 slikovnih pik in velikostjo 11.1 MB;
- DOCX s 155 stranmi in velikostjo 1.51 MB;
- PDF s 24 stranmi in velikostjo 0.99 MB;
- PPT s 341 diapozitivi in velikostjo 3.98 MB;
- MP3 z dolžino 16 minut in 15 sekund, velikostjo 25.7 MB in povprečno bitno hitrostjo 221 Kbps;
- MP4 z 29.99 sličic na sekundo in dolžino 2 minuti in 37 sekund. Testirali smo tri enake posnetke, ki so se razlikovali le v številu slikovnih pik (1080, 1440, 2160).

Rezultati

Google Drive je datoteko formata JPG prikazal hitro, a na sliki so bili vidni artefakti kot posledica kompresije. Pri datotekah formatov DOCX, PDF in PPT je hipno prikazal le prvo stran, za vse naslednje pa je potreboval nekaj sekund. Za začetek predvajanja skladbe formata MP3 je potreboval okoli 5 sekund in skladba se je po naših občutkih predvajala brez motenj v zvoku. Pri odpiranju datotek MP4 smo opazili, da ne zmore prikazati resolucije večje od 1080 slikovnih pik, saj je datoteki z višjima resolucijama skrčil na to velikost. Pojavili so se zelo moteči kompresijski artefakti.

Mega je za odpiranje datoteke JPG porabila slabo sekundo in pri tem ponudila nekoliko zrnato sliko. Odpiranja datotek DOCX, PDF, PPT, MP3 in MP4 ne podpira.

Dropbox je datoteko formata JPG prikazal hitro in brez vidnih artefaktov. Tudi pri datotekah formatov DOCX in PDF nismo dolgo čakali, da smo zagledali najprej prvi dve strani, nato pa še celoten dokument. Težave pa je imel pri datoteki formata PPT, saj tudi po minuti čakanja datoteke ni uspel prikazati. Pri tem ni javil nobenega sporočila o morebitni napaki. Ob kliku na datoteko formata MP3 se je po slabih dveh sekundah odprl Dropboxov predvajalnik glasbe, ki je za samo predvajanje datoteke potreboval še dodaten klik na gumb. Video predvajalnik se je ob datotekah MP4 odprl v trenutku, ampak je bila slika pri vseh posnetkih kompresirana in zelo moteča.

6.7 Zaključek

Po izvajanju meritev, smo začeli tudi na spletu iskati podobne eksperimente. Dobili smo dve analizi oblačnih ponudnikov. Prva [71] je bila bolj tehnična, opazovali so vpliv koščkov, hitrost prenosa pri hkratnem pošiljanju večih datotek in kompresijo. Ugotovili so da sta Google Drive in Dropbox kar enakovredna v zmogljivosti, Google zaradi svoje obstoječe internetne infrastrukture in Dropbox zaradi svoje hitrosti pri sinhronizaciji podatkov.

Druga [75] podobna analiza pa ni bila tako tehnična. Pošiljali/prenašali so podatke in jih, če so lahko, tudi urejali. So pa med sinhronizacijo opazovali kaj se zgodi, če jo za nekaj časa prekineš. Pri njihovem testiranju je zanimivo, da so testiranje izvajali tudi iz geografsko različnih lokacij (San Francisco, Dublin, Tokio). Pri vseh njihovih testih, se je Dropbox for Business boljše odrezal kot preostali.

Z eksperimentom dostopnosti 6.5.1 in eksperimentom 24-urnega nalaganja datotek 6.5.4 smo odkrili, kateri del dneva so strežniki ponudnikov najbolj obremenjeni, in sicer je to med 2. in 4. ter 7. in 8. uro zjutraj.

Iz rezultatov opravljenih eksperimentov smo spoznali, da na čas, potreben za nalaganje ali prenašanje datoteke, lokacija postavljenih strežnikov nima tako velikega vpliva kot smo prvotno mislili. Oblačni strežniki nas kot brezplačne uporabnike namreč dodelijo tja, kamor jim najbolj ustreza, četudi so strežniki do nas zelo oddaljeni. Pri uporabi brezplačnega načina hranjenja podatkov, si oblačni ponudniki dovolijo tudi poljubno zmanjšati pasovno širino, kar se je pokazalo pri eksperimentu 6.5.2.

Eksperiment 6.5.2 je pokazal, da datoteke različnih velikosti najhitreje nalaga Dropbox, prenaša pa Google Drive. Pri nalaganju in prenašanju več enakih datotek pa se je pri obeh načinih prenašanja podatkov najbolje odrezal Google Drive (glej eksperiment 6.5.3).

S testiranjem uporabniške izkušnje v razdelku 6.6 smo pridobili nekaj ključnih in pomembnih informacij za uporabnika spletne različice oblačnih ponudnikov. Najslabši vtis je pustila Mega, ki izmed vseh testiranih formatov datotek odpira le slike JPG in še te preoblikuje v zrnato obliko. Če uporabniku ni po-

membno, da si lahko ogleduje datoteke v brskalniku, ampak mu je pomembna le količina brezplačnega prostora, je Mega s svojimi 50 GB najboljša izbira izmed vseh treh ponudnikov. Velja omeniti še, da smo med nalaganjem datotek za testiranje uporabniške izkušnje opazili še, da se prenos pri velikih datotekah kdaj ustavi. Na drugi strani tako Google Drive kot Dropbox nimata problema z različnimi formati datotek. Prvi zna nekoliko bolje ravnati z datotekami PPT, drugi pa JPG. Oba sta pri predvajanju videoposnetkov sliko kompresirala.

Poglavje 7

Analiza zmogljivosti oblačnie storitve Heroku

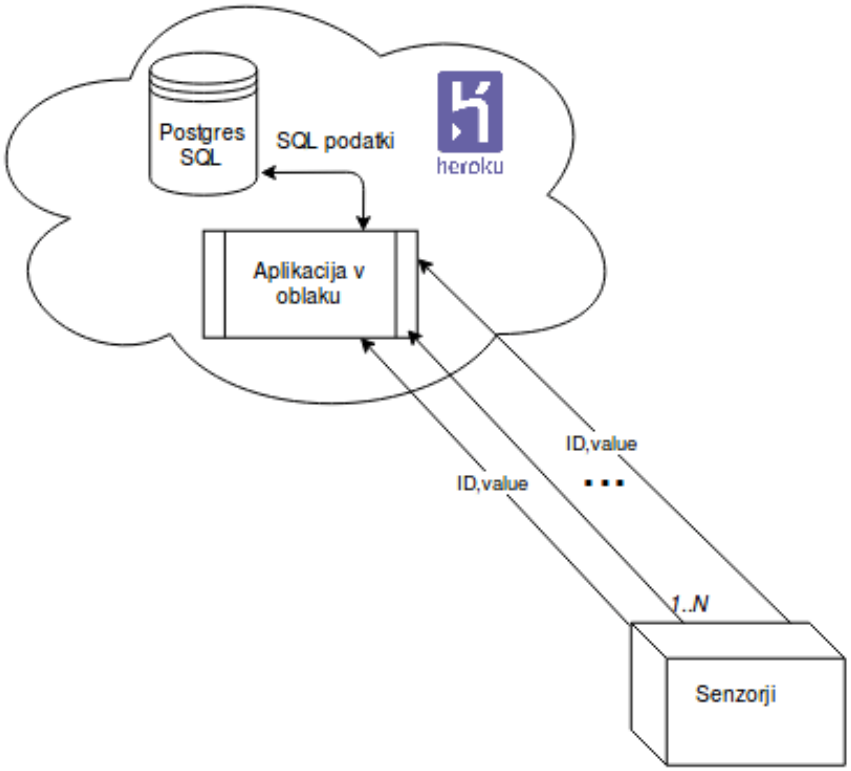
Anže Dežman, Edo Ljubijankić, Maks Vrščaj, Aljaž Markežič

7.1 Opis problema

V današnjem času imamo na voljo ogromno oblačnih storitev, ki nam ponujajo okolje za razvijanje in zaganjanje aplikacij, ki so lahko napisane v različnih jezikih (Javascript, Ruby, Python,...). Oblačnim storitvam takega tipa se reče "okolje kot storitev", oziroma v angleškem izvirniku"Platform as a sistem - PaaS". Namen seminarske naloge je testirati in analizirati zmogljivosti ene izmed PaaS oblačnih storitev po imenu Heroku [76].

Heroku se od konkurence razlikuje po tem da podpira velik izbor programskih jezikov (Java, Python, PHP, Node.js, Scala,...), med katerimi so tudi manj popularni jeziki kot Scala ali pa Groovy, ki ga večina oblačnih storitev ne podpira. Zanima nas koliko nam oblačna storitev ponuja na področju zmogljivosti, kolikšna je razlika med oglaševano zmogljivostjo in dejansko, ali se bolj splača razvijati aplikacijo lokalno ali je ponujena brezplačna storitev dovolj zmogljiva za naše standarde itd.
Na vsa ta in mnoga druga vprašanja, ki se nam bodo pojavila tekom testiranja ter kakšne metode testiranja bomo uporabili, bomo odgovorili v naslednjih razdelkih.

Ker je storitev Heroku namenjena razvijanju, gostovanju in izvajanju spletnih aplikacij, smo se osredotočili na testiranje izvajanja neke določene aplikacije.



Slika 7.1: Shema aplikacije, ki jo testiramo.

Znotraj oblaka smo realizirali strežniški del aplikacije, ki sprejema podatke iz N senzorjev izven oblaka in jih shrani v podatkovno bazo. Na sliki 7.1 je shema zgoraj opisane aplikacije.

7.2 Izbira tehnologij

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

7.2.1 Node.js

Node.js [77] je odprtokodno orodje, ki z okoljem v programskem jeziku JavaScript služi kot strežniška aplikacija oz. orodje. Zaradi dogodkovne odzivnosti omogoča razvoj hitrih spletnih strežnikov. Razvijalci lahko ustvarijo zelo skalabilne strežnike brez uporabe niti, temveč s poenostavljenim modelom dogodkovnega programiranja, ki uporablja funkcije povratnega klica (angl. *callback*

functions), s katerimi signalizira končanje opravila. Spletni strežnik se ustvari zelo enostavno, omogoča pa še vrsto drugih funkcionalnosti kot so I/O operacije datotečnega sistema, mrežne storitve (DNS,HTTP,TCP,TLS/SSL, UDP), kriptografske funkcije, podatkovne tokove itd. Node.js moduli so narejeni tako, da zmanjšajo kompleksnost pisanja strežniških aplikacij.

7.2.2 PostgresSQL

PostgresSQL, v nadaljevanju imenovan Postgres, je odprtokodna objektno-relacijska podatkovna baza. Izdelana je kot poslovna aplikacija, zato podpira veliko naprednih funkcij, med njimi tudi Multi-Version Concurrency Control (MVCC), asinhrono replikacijo itd. [78] Je edina podatkovna baza, ki jo okolje Heroku podpira "iz škatle". Možno jo je aktivirati kot dodatek za vsako Heroku aplikacijo. Za zastonjske račune ima omejitev največ 10000 zapisov v celotni podatkovni bazi [79]. Za vse uporabnike podpira neadministratorske SQL ukaze, ki jih lahko vnesemo preko ukazne vrstice. Podatkovno bazo je z aplikacije možno prekopirati na lokalni računalnik ali pa do nje dostopati iz drugih aplikacij. Do nje se lahko povežemo s katerimkoli jezikom, ki ga Heroku ponuja [80].

7.3 Scenarij testiranja

7.3.1 Aplikacija v oblaku

Aplikacijo v oblaku smo realizirali z Node.js. Aplikacija ima stalno odprto povezavo do PostgresSQL podatkovne baze.

Omogoča način posamične posodobitve senzorjev, kjer se stanje posameznega senzorja posodobi posamično, ko aplikacija podatke o stanju senzorja prejme. Na strani aplikacije se izračunajo časi potovanja podatkov senzorja in časi vstavljanja teh podatkov v podatkovno bazo.

Podatki, ki jih aplikacija prejme od posameznega senzorja imajo strukturo *[številka senzorja (ID), stanje senzorja (value), čas pošiljanja]*. Enačbi 7.1 in 7.2 se uporabita za izračun *vrstice* in *stolpca* v tabeli v podatkovni bazi, kjer se bo *stanje senzorja* posodobilo. V obeh enačbah S predstavlja število stolpcev tabele v podatkovni bazi.

$$vrstica = ID/S, \tag{7.1}$$

$$stolpec = IDmodS. \tag{7.2}$$

V podatkovno bazo se shrani samo *stanje senzorja*, podatki štoparice pa se hranijo v aplikaciji.

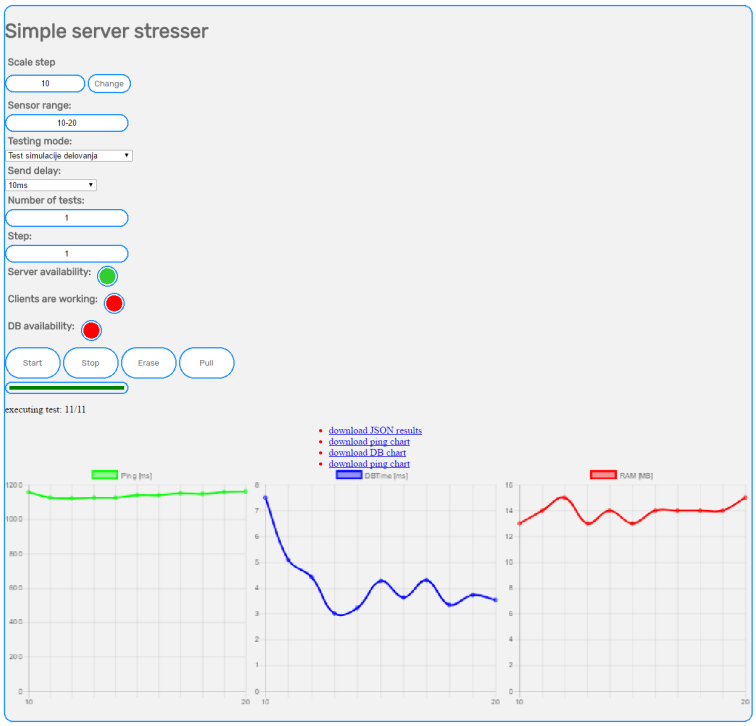
7.3.2 Podatkovna baza

Postavitev same podatkovne baze je preprosta, saj se aplikaciji doda samo dodatek (Add-on) *Heroku Postgres* iz ukazne vrstice ali iz administratorkse plošče

aplikacije [81].

Kot je bilo že omenjeno v razdelku 7.2.2 je zastonjska različica omejena na 10000 zapisov v podatkovni bazi, zato nameravamo tabele uporabiti kot matrike, naprimer tabela z 1000 zapisi in 1000 stolpci lahko hrani 1000000 podatkov. V resnici je stolpcev 1001, saj se prvi rabi za identifikacijo vrstice. Na tak način bo možno uporabiti več tabel.

Izgradnja tabele se je zgodila samo enkrat, uporabljalo pa se je 1 do N celic, ki so se posodabljale.



Slika 7.2: Uporabniški vmesnik simulatorja senzorjev.

7.3.3 Simulator senzorjev

Simulator senzorjev se nahaja izven oblaka. S tem se simulira oddaljenost senzorjev od aplikacije, lastnost, ki bi jo senzorji imeli v realnem svetu.

Realiziran je kot spletna stran, na kateri se nastavi razmak števila aktivnih senzorjev, časovni razmak (*delay*) med pošiljanji podatkov posameznega senzorja, korak, za katerega se poveča število aktivnih senzorjev za naslednje testiranje in število ponovitev testov.

Omogoča način testiranja pošiljanja podatkov iz simulatorja senzorjev na spletno aplikacijo in način testiranja samo vstavljanja v podatkovno bazo. Proces testiranja se aktivira s strani simulatorja senzorjev.

Delay [ms]	Povp. čas [s]
10	20.90 ± 3.19
100	19.63 ± 1.74
500	19.98 ± 1.85
1000	19.99 ± 1.60
2000	20.34 ± 1.68

Tabela 7.1: Tabela prikazuje povprečne čase potovanja zahtev od senzorjev do aplikacije za celotno vrsto senzorjev.

Zaradi lažjega dostopa do simulatorja senzorjev smo spletno stran naložili na Heroku in je dostopna na spletnem naslovu `https://zzrs.herokuapp.com`. Spletna stran se prenese v brskalnik in ker je JavaScript originalno namenjen izvajanju na strani brskalnika, se lastnost oddaljenosti simulatorja senzorjev od oblaka spletne aplikacije ne izgubi.

Simulator senzorjev simulira N senzorjev, ki aplikaciji v oblaku pošljejo svojo *številko senzorja* (*ID*), svoje *stanje senzorja* (*value*) in čas začetka pošiljanja teh podatkov. V simulatorju senzorjev se shranjevanje časovnih podatkov ne izvaja, omogočeno pa je pridobivanje rezultatov za posamična testiranja, ki jih prikaže kot grafe. Slika 7.2 prikazuje izgled uporabniškega vmesnika simulatorja senzorjev

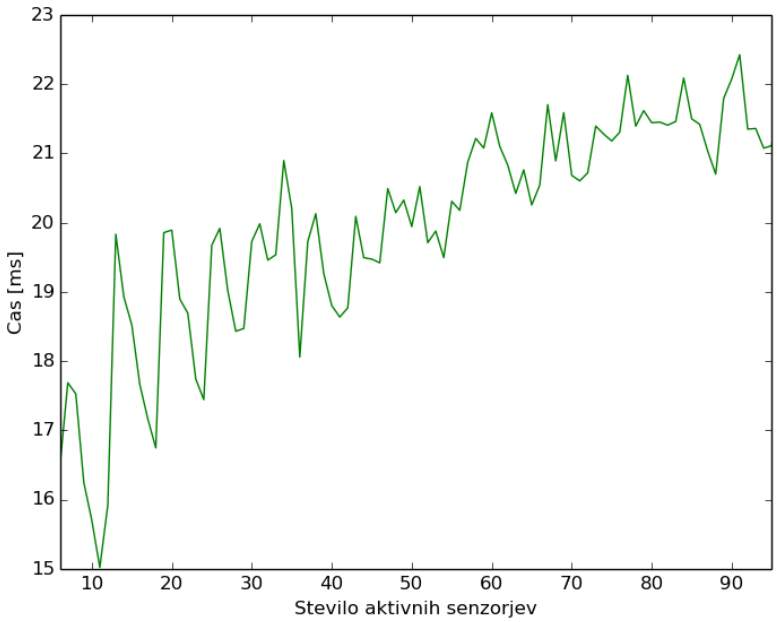
7.4 Testiranje

7.4.1 Eksperiment 1: Čakalna vrsta na spletni aplikaciji

Ker je spletna aplikacija napisana v Node.js, ki uporablja jezik JavaScript, ki ne podpira paralelizma, obstaja velika verjetnost, da se zahteve za vstavljanje podatkov v podatkovno bazo ustavijo v čakalni vrsti JavaScript-a. Postavi se hipoteza, da je ozko grlo aplikacija. Za potrebe testa se simulator senzorjev spremeni tako, da se zaporedoma pošilja vedno večjo vrsto senzorjev oz. njihovih zahtev.

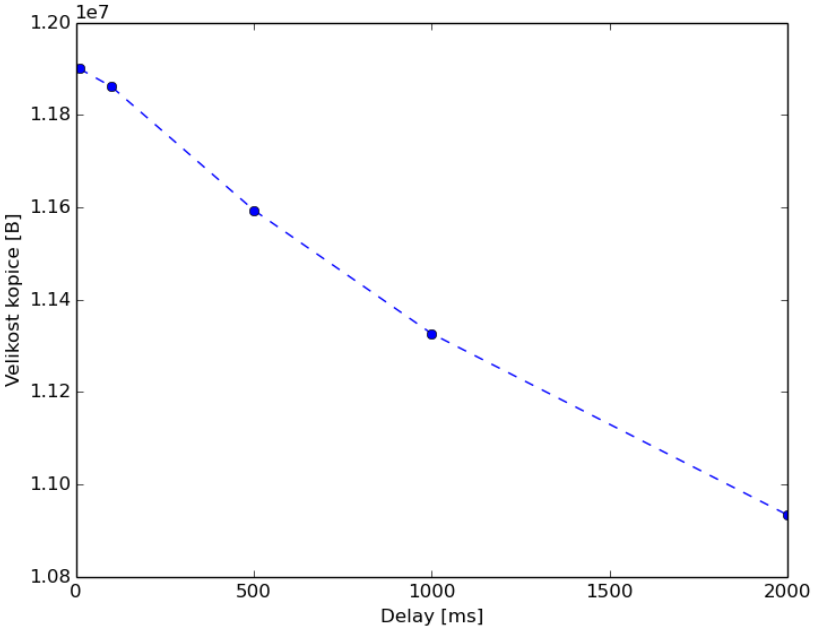
Opis poteka testa: Začenši z vrsto zahtev 5 senzorjev, med katerimi je razmak med zahtevami definiran s spremenljivko "delay", se v naslednjem koraku pošlje vrsta 6 zaporednih senzorjev, v naslednji iteraciji 7 zaporednih zahtev, itn. Vrsta se je v tem testu povečevala do 100 zaporednih zahtev senzorjev. Vsak test je bil izveden petkrat, za vsako spremeljivko "delay", ki je bila nastavljena na 10, 100, 500, 1000 ter 2000 milisekund. Tako je bilo skupno število testov 25, kar nanese da je skupno število poslanih vrst zahtev 25 * 95 = 2375.

Pri vsakem testu se je merilo čas prejetja vsake zahteve na aplikacijo kot tudi porabo pomnilnika. Tabelarični rezultati 7.1 vsebujejo tudi st. odklon, s čimer lahko vidimo razpršenost testiranja. Vidimo lahko, da povprečni čas potovanja zahtev vseh vrst(torej od 5 do 100) ni odvisen od nastavljenega delaya. Se pa bo v naslednjem poglavju izkazalo, da pri večjem številu testov z manjšimi delayi to ne drži in da se pri delayu 5ms vsi časi znižajo pod cca 50ms. Je pa povprečni čas potovanja zahtev v tem testu odvisen od velikosti vrste, kot lahko vidimo spodaj.



Slika 7.3: Povprecni časi 5 testiranj za delay 1000ms.

Pričakovali bi, da se v skladu z našo hipotezo o ozkem grlu na aplikaciji s povečevanjem vrste, kot prikazuje slika 7.3 povečuje tudi čakalna vrsta zahtev, kar naj bi bilo razvidno v vedno večjem povprečnem času potovanja zahteve od senzorja do aplikacije. Kar lahko najprej opazimo je, da se rezultati meritev skladajo s to trditvijo, saj se časi z večanjem velikosti vrste povečujejo.



Slika 7.4: Povprečna poraba pomnilnika glede na nastavljen "delay".

Tudi če pogledamo na porabo pomnilnika, se le-ta zmanjšuje pri vedno večjem razmaku med poslanimi zahtevami, kar kaže na to, da se pri krajših oz. prehitrih razmakih ustvarja vrsta zahtev na aplikaciji, ki se na grafu pozna v večji porabi pomnilnika. S tem tudi sprejmemo oz. dokažemo našo hipotezo.

7.4.2 Eksperiment 2: Čakalna vrsta na podatkovni bazi

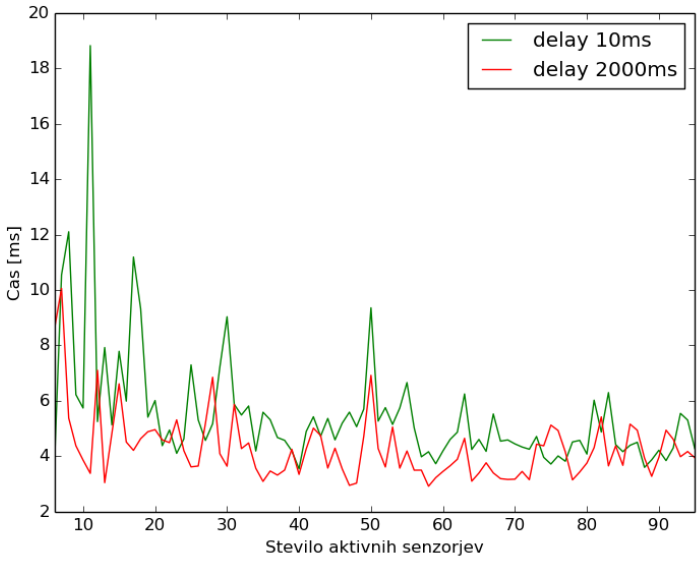
S tem testom želimo preveriti, ali je ozko grlo naše aplikacije podatkovna baza. Postavi se hipoteza, da se čakalna vrsta ustvarja pri zapisu podatkov senzorjev v podatkovno bazo. Test je bil izveden na enak način kot pri prejšnjem eksperimentu za delay 10ms ter 2000ms, le da tokrat merimo čas zapisa senzorja v podatkovno bazo. Test je bil za 10ms in 2000ms izveden petkrat, skupaj torej 10 testov, kar nanese 10*95vrst=950 vrst senzorjev.

Rezultati meritev:

Iz tabele 7.2 je razvidno, da so povprečni časi vnosa v podatkovno bazo podobni pri obeh, kar kaže na to, da se vrsta ne generira zaradi manjšega delaya. Kljub temu, da je meritev pri 10ms razmaka v povprečju za dobro milisekundo počasnejša, ne moremo trditi da je to zaradi nastajanja vrste pri zapisu v podatkovno bazo, saj je disperzija velika dobri 2 milisekundi. Tako ne vemo ali je razlika zgolj zaradi šuma v podatkih ali se dejansko ustvarja vrsta.

Delay [ms]	Povp. čas [s]
10	5.41 ± 2.12
2000	4.27 ± 1.16

Tabela 7.2: Tabela prikazuje povprečne čase vnosa podatka senzorja v podatkovno bazo.



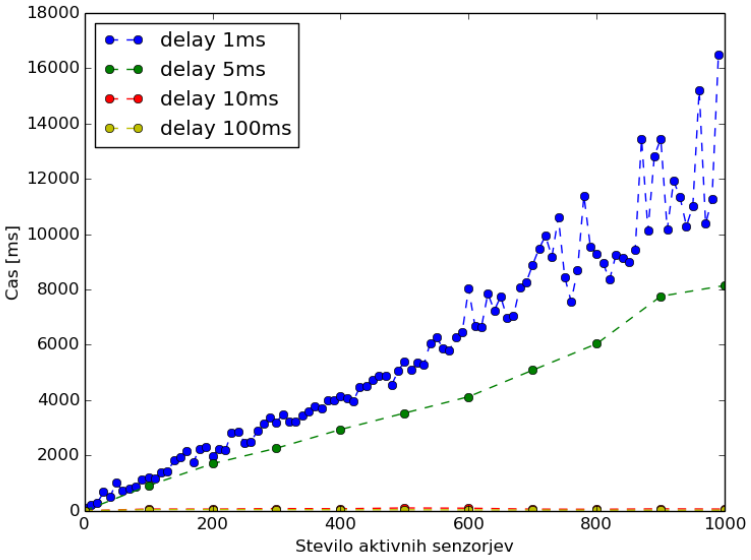
Slika 7.5: Čas vnosa v podatkovno bazo v odvisnosti od velikosti vrste za dva različna delaya.

Če primerjamo ta graf z grafom iz prejšnjega eksperimenta 7.3 vidimo, da se tu ne ustvarja latenca z večanjem števila aktivnih senzorjev. Lahko trdimo, da je latenca pri vnosu v podatkovno bazo konstantna in neodvisna od števila poslanih zahtev, medtem ko pa je latenca potovanja zahtev na aplikacijo odvisna od števila poslanih zahtev. S tem lahko zakjučimo, da hipoteza o ozkem grlu na podatkovni bazi ne velja.

7.4.3 Eksperiment 3: Čakalna vrsta na spletni aplikaciji z večjim številom vrst senzorjev oz. testov

Vrsta se je v tem testu povečevala do 1000 zaporednih zahtev senzorjev. Vsak test je bil izveden petkrat, za vsako spremeljivko "delay", ki je bila nastavljena na 1, 5, 10 in 100ms. Tako je bilo skupno število testov 20. Za delay 1ms smo test izvedli z večanjem velikosti vrste po koraku deset od velikosti 10 do

1000, kar nanese na 100 poslanih vrst. Pri vrstah nad 600 zahtev nanese zelo dolge latence pošiljanja in je test za 1ms delay trajal dobri 2 uri, zato smo pri naslednjih delayih povečali korak na 100. Rezultati meritev se nahajajo na spodnjem grafu 7.6, kjer lahko vidimo največje latence pri delayu 1ms. Na grafu vsaka pika označuje povprečje 5 testiranj. Kot rečeno je graf za 1ms izjema in vsebuje povečevanje koraka vrste zahtev 10, ostali pa 100.

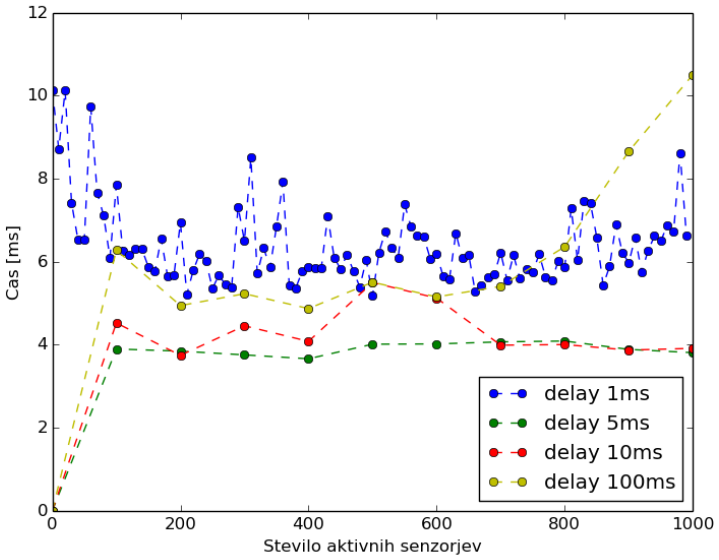


Slika 7.6: Povprečni čas potovanja zahtev za različne delaye, pri čemer povečujemo velikost vrste od 10 do 1000. Za delay 1ms je korak povečevanja 10, pri ostalih delayih pa zaradi dolgega izvaja testa povečamo korak na 100.

Slika 7.6 potrjuje eksperiment št. 1 in njegovo hipotezo, saj lahko tudi tukaj razberemo, da je z večanjem vrste senzorjev tudi večji delay. Vendar kar pa na novo opazimo je, da obstaja meja med zelo veliko latenco ter „zanemarljivo“ latenco. Če pogledamo delaya za 1ms in 5 ms vidimo, da znaša latenca pri vrsti 500 senzorjev dobre 4 oz. 5 sekund, medtem ko pri delayih 10ms in 100ms ne preseže nobeden test niti 100ms. Graf iz eksperimenta št. 1 prikazuje, da tudi slednja testa vsebujeta linearno rast glede na število aktivnih senzorjev, kar je tukaj popačeno in zgleda konstantno, saj je skala velikosti grafa po času velika zaradi bistveno večjih delayov za 1ms in 5ms. Zato testov nad 100ms delaya nismo opravljali več, saj so latence minimalne. S tem lahko popravimo zaključek našega prvega eksperimenta tako, da je povprečni čas potovanja zahtev vrst (oz. povečevanje čakalne vrste zahtev) odvisen od nastavljenega delaya in ni odvisen samo od večanja vrste aktivnih senzorjev.

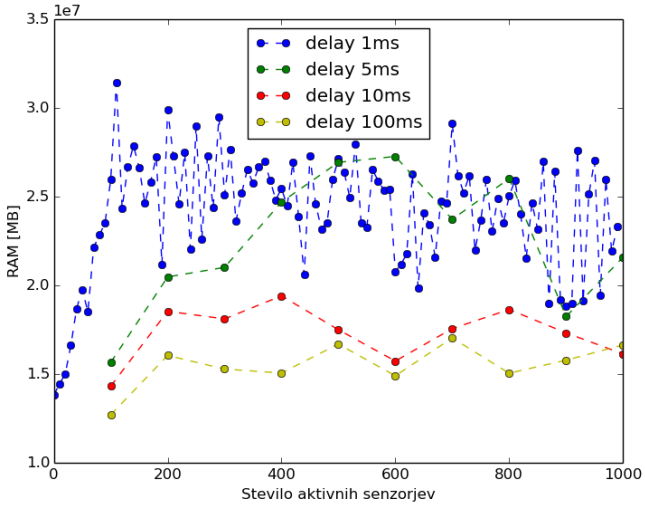
Slika 7.7 prikazuje graf časov vnosa v podatkovno bazo in kot lahko vidimo je le-ta konstanten pri vseh delayih, če zanemarimo minimalna odstopanja zaradi

šuma pri meritvah. Konstantno latenco oz. neodvisnost latence od velikosti vrste lahko argumentiramo s podatki iz prejšnje meritve, kjer lahko vidimo, da so časi latence nastale vrste na aplikaciji bistveno večji kot je povprečni čas vnosa v podatkovno bazo. Npr. pri 1ms delay za vrsto 600 senzorjev je latenca kar 100krat manjša in je tako zanemarljiva v primerjavi z latenco zaradi povečevanja čakalne vrste zahtev na aplikaciji.



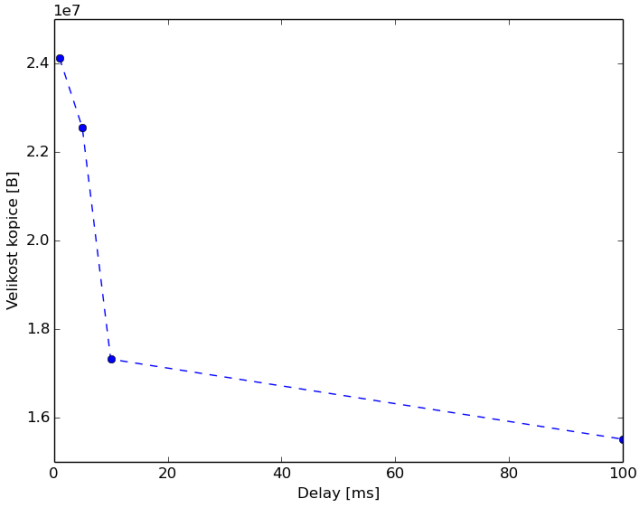
Slika 7.7: Čas vnosa v podatkovno bazo v odvisnosti od števila aktivnih senzorjev.

Slika 7.8 prikazuje velikost kopice v odvisnosti od števila aktivnih senzorjev. Lahko vidimo, da je konstantna poraba pomnilnika za vsak delay. Vendar kar je zares pomembno tukaj je naša prva hipoteza o čakalni vrsti zahtev na aplikaciji, ki se vidi tudi v manjši porabi pomnilnika za večji delay med zahtevami.



Slika 7.8: Velikost RAM-a v odvisnosti od števila aktivnih senzorjev.

Manjšo porabo pomnilnika iz slike 7.8 prikazuje na bolj razviden način slika 7.9, ki povpreči vrednosti iz slike 7.8 za vsak delay in lepo prikazuje pričakovano manjšo porabo pomnilnika pri večjih delayih.



Slika 7.9: Velikost kopice na RAM-u v odvisnosti od nastavljenega delaya.

7.5 Zaključek

V našem delu smo testirali zmogljivost aplikacije, ki je pošiljala podatke senzorja v podatkovno bazo. Testne scenarije smo postavili tako da smo ocenjevali čase potovanja podatkov senzorja in čase vstavljanja teh podatkov v podatkovno bazo. Našo hipotezo, da se čakalna vrsta ustvarja na spletni aplikaciji smo potrdili z eksperimentoma v razdelkih 7.4.1 in 7.4.3.

Glede na rezultate iz slik 7.6 in 7.7 je razvidno, da naraščanje čakalne vrste izgine, če *delay* med pošiljanji podatkov senzorjev nastavimo na 10 milisekund ali več. Če je *delay* nastavljen na 10 milisekund je iz slike 7.7 razvidno, da *Postgress* porabi 4–6 milisekund da podatek vstavi v bazo. To pomeni, da je spletna aplikacija zmožna prejeti približno 6000 podatkov senzorjev na minuto medtem ko bi bil *Postgress* za isti *delay* zmožen vnesti do 15.000 podatkov na minuto. Glavna omejitev spletne aplikacije je uporaba *Node.js*, ki ne podpira paralelne obdelave zahtev, ki bi pripomogla k hitrejšemu sprejemu podatkov iz senzorjev. Zmogljivost spletne aplikacije bi se izboljšala, če bi jo prepisali v jezik, ki podpira paralelno izvajanje.

Če bi aplikacija bila spisana v jeziku, ki podpira paralelno izvajanje, bi jo lahko uporabili v električnih merilnih napravah kot jih ima podjetje Iskraemeco. Iskraemeco, d.d. je eden izmed vodilnih ponudnikov na področju merilne tehnologije [82]. S števci električne energije in ostalimi izdelki pametnega merjenja omogočajo energetskim podjetjem učinkovito upravljanje z energijo ter jim pomagajo oblikovati prihodnost na področju upravljanja z energijo.

Poglavje 8

Analiza zmogljivosti oblačne storitve DigitalOcean

Žiga Kokelj, Tadej Hiti,
Miha Bizjak, Matej Kristan

8.1 Opis problema

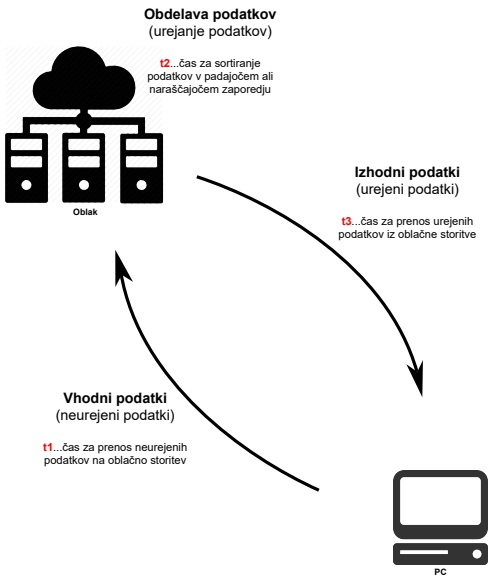
V zadnjem desetletju se na vseh področjih računalništva vse bolj uveljavljajo oblačne storitve in koncept odjemalec - strežnik (angl. *client - server*). Z večanjem hitrosti internetnih povezav končne delovne točke (predvsem osebni računalniki) izgubljajo del svojih primarnih funkcij. Hranjenje in obdelavo podatkov vse bolj prepuščajo oblačnim storitvam. V naši analizi smo se osredotočili na obdelavo podatkov na strani strežnika. Za breme smo izbrali datoteke različnih velikosti, ki vsebujejo naključna števila. Te datoteke odjemalci pošljejo na strežnik, ta pa jih uredi v naraščajočem vrstnem redu in pošlje nazaj. Na sliki 8.1 je grafičen prikaz opisanega problema. Naše testiranje bo obsegalo merjenje časov čakanja odjemalcev na urejeno datoteko ter merjenje obremenjenosti strežnika.

8.2 Izbira tehnologij in algoritmov

V tem razdelku so opisane tehnologije in algoritmi, ki smo jih uporabili za našo analizo.

8.2.1 Tehnologija na strani strežnika

Na oblačni storitvi smo implementirali strežnik, ki je realiziran v jeziku JavaScript [83] z uporabo knjižnice Node.js [84]. Strežnik čaka na HTTP zahteve s



Slika 8.1: Shema delovanja sistema.

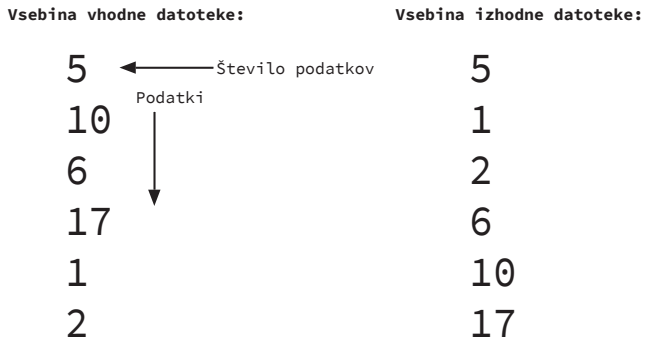
strani klienta, kateri pošlje datoteko števil za urejanje. Ob sprejemu datoteke s strani strežnika se začne izvajati program, ki števila v datoteki uredi v naraščajočem vrstnem redu. Ker je za namene testiranja oblačne infrastrukture bistveno, da se program čim hitreje izvrši smo za ta namen uporabili programski jezik C [85], ki spada med najhitrejše programske jezike, zaradi njegove nizko-nivojske lege na procesorski arhitekturi. Po končanem sortiranju strežnik urejeno datoteko pošlje nazaj odjemalcu.

8.2.2 Tehnologija za avtomatizacijo odjemalcev

Zaradi avtomatskega testiranja smo napisali skripto v programskem jeziku Python [86], ki omogoča avtomatsko pošiljanje datotek na strežnik, ter čaka na odgovor (urejeno datoteko s števili). Ob tem zabeležimo še čas pred pošiljanjem zahteve in čas po prejetju urejene datoteke, da dobimo celotni čas potreben za pošiljanje in sortiranje na strežniku. Ker je odjemalcev lahko večje število, smo ta problem rešili z nitmi, kjer vsaka nit predstavlja enega odjemalca in pošilja zahteve na strežnik preko istega IP naslova.

8.2.3 Generiranje vhodnih podatkov

V programskem jeziku C smo napisali generator datotek, ki ustvari datoteko željene velikosti z naključnimi števili. Za teste smo generirali datoteke velikosti 10000, 20000, 30000, 40000 in 50000 števil tipa integer.



Slika 8.2: Slika prikazuje primer datoteke s katero operirata strežnik in odjemalec.

8.2.4 Izbrani algoritmi za sortiranje

Za namen testiranja smo implementirali dva algoritma. Najprej smo testirali z algoritmom za mehurčno urejanje (ang. *bubble sort*), ki ima časovno zahevnost $O(n^2)$. Da bi testirali vpliv izbire algoritma smo tudi implemenirali hitrejši algoritem urejanja s porazdelitvami (ang. *quicksort*), kateri ima povprečno časovno zahtevnost $O(n * log(n))$.

Listing 8.1: Bubblesort

```
function bubblesort(int[] data){
   for(int i = 0; i <
       data.length; i++){
      for(int j = 0; j <
          data.length; j++){
         if(data[i] < data[j]){
            int tmp = data[i];
            data[i] = data[j];
            data[j] = tmp;
         }
      }
   }
}
```

Listing 8.2: Quicksort

```
function quicksort(int[] data,
    int lo, int hi){
   if(lo < hi){
      int p = partition(data,
          lo, hi);
      quicksort(data, lo, p-1);
      quicksort(data, p+1, hi);
   }
}

function partition(int[] data,
    int lo, int hi){
   int pivot = data[hi];
   int i = lo -1;
   for(int j = lo; j < hi; j++){
      if(data[j] <= pivot){
         i = i+1;
         if(i != j){
            int tmp = data[i];
            data[i] = data[j];
            data[j] = tmp;
         }
      }
   }
   int tmp = data[i+1];
   data[i+1] = data[j];
   data[j] = tmp;
   return i+1;
}
```

8.3 Izbira ponudnikov

Najprej smo izbrali ponudnika oblačnih storitev Cloud9, ki pa ni najbolj primeren za naš problem, saj je v prvi vrsti razvojno okolje na spletu in se ne osredotoča na ponujanje strežniške infrastrukture. Pri računsko zahtevnejših testih smo naleteli na težave, saj je strežnik po 120 sekundah povezavo zaprl. To je bil eden glavnih razlogov, da smo se odločili za zamenjavo ponudnika oblačnih storitev, saj smo le tako lahko izvedli vse željene teste. Izbrali smo si ameriško podjetje DigitalOcean, ki je ponudnik istoimenske oblačne infrastrukture. Na našo srečo sodelujejo v projektu Github Education Pack in tako študentom nudijo 50$ kredita za uporabo njihovih storitev.

Svoje strežnike imajo postavljene na lokacijah naštetih v nadaljevanju:

- New York (ZDA),
- San Francisco (ZDA),
- Amsterdam (Nizozemska),
- Singapur (Singapur),
- London (Združeno kraljestvo),
- Frankfurt (Nemčija),
- Bangalore (Indija).

V okviru študentskih kreditov so nam na voljo tri različne konfiguracije, ki se med seboj razlikujejo po količini dodeljenih resursov. Podatki o konfiguracijah so na voljo v tabeli 8.1.

RAM	Stevilo procesorjev	Prostor na disku (SSD)	Cena na mesec
512 MB	1	20 GB	5$
1 GB	1	30 GB	10$
2 GB	2	40 GB	20$

Tabela 8.1: Konfiguracije strežnikov, ki so na voljo za študentske kredite.

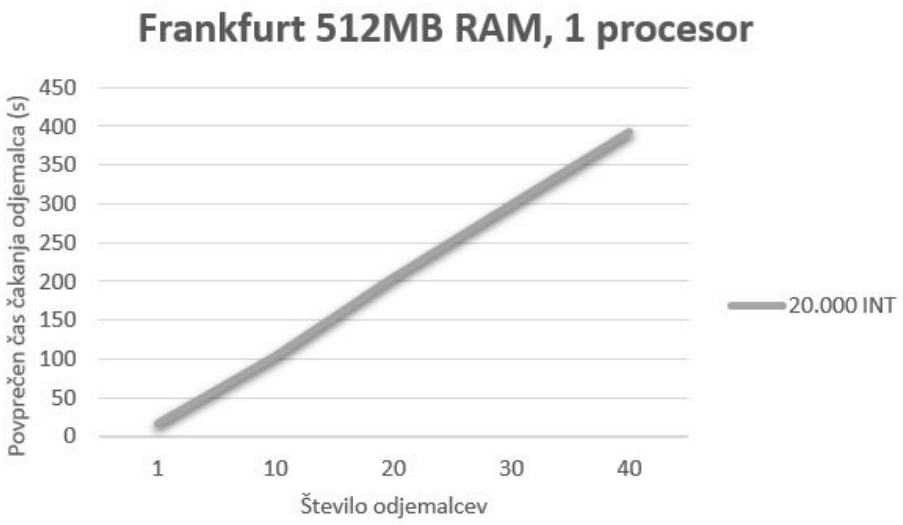
8.4 Rezultati meritev

8.4.1 Število odjemalcev - Eksperiment 1

- **Hipoteza:**
  Naša hipoteza je, da se povprečen čas čakanja odjemalca približno linearno povečuje z večanjem števila odjemalcev, ki hkrati pošiljajo datoteke na strežnik.

- **Okoliščine meritve:**
  Testiranje smo izvedli v četrtek 18.5.2017 med 21:00 in 21:30 na strežniku v Frankfurtu. Na strežniku je tekel operacijski sistem Linux Ubuntu 16.04. Na voljo smo imeli 512 MB RAM-a, 1 procesor ter 20 GB prostora na SSD trdem disku.
  Na strežniku je bil izbran algoritem za mehurčno urejanje (ang. *bubble sort*). Skripto za simulacijo odjemalcev smo pognali iz študentskega naselja Rožna dolina (Ljubljana). Za dostop do interneta je bila uporabljena internetna povezana s hitrostjo 100/100 Mbps.
  Na strežnik so klienti (niti) pošiljali datoteke, ki so vsebovale 20.000 števil tipa integer in merili čase pri 1, 10, 20, 30 in 40 odjemalcih.

- **Rezultati meritve:**
  Rezultati meritve so vidni na sliki 8.3.



Slika 8.3: Graf povprečnega časa čakanja odjemalcev v odvisnosti od števila odjemalcev.

Število odjemalcev	Povprečni cas obdelave [s]	Standardna deviacija
1	16,06	0
10	103,79	1,63
20	205,44	4,88
30	300,89	12,83
40	393,28	14,60

Tabela 8.2: Tabela časov obdelav datoteke z 20 tisoč celimi števili.

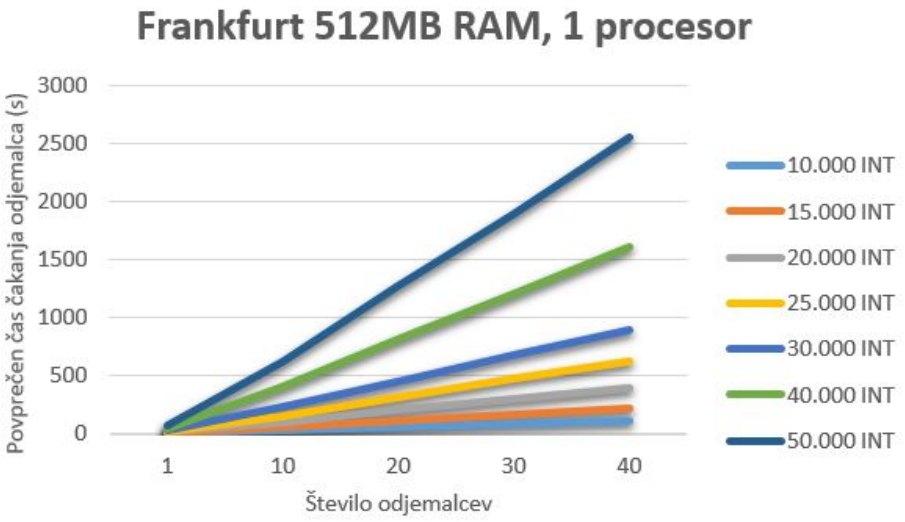
- **Komentar meritve:**
  Iz slike 8.3 in tabele 8.2 lahko zelo razločno vidimo, da se z večanjem števila odjemalcev linearno povečuje tudi povprečen čas čakanja posameznega odjemalca. Opažen rezultat pričakovano sovpada z našo hipotezo, saj ima strežnik omejeno število resursov in morajo zato ob večjem številu zahtevkov odjemalci čakati dlje. Prav tako lahko v tabeli opazimo, da se z večanjem števila odjemalcev povečuje tudi standardna deviacija, kar pa pomeni, da ob višjih obremenitvah (več odjemalcev) vedno težje predvidimo čas obdelave datotek določenega odjemalca, saj prihaja do vedno večjih odstopanj.

8.4.2 Velikost datotek - Eksperiment 2

- **Hipoteza:**
  S sledečim eksperimentom smo želeli preveriti odvisnost povprečnega časa čakanja klientov glede na velikost datotek. Da bi izničili vpliv števila odjemalcev na ta poizkus smo teste ponovili na več različnih številih odjemalcev. V tem primeru predpostavljamo hipotezo, da se z večanjem velikosti datoteke veča tudi povprečni čas čakanja klientov.

- **Okoliščine meritve:**
  Testiranje smo izvedli v četrtek 18.5.2017 med 21:30 in 24:00 na strežniku v Frankfurtu. Na strežniku je tekel operacijski sistem Linux Ubuntu 16.04. Na voljo smo imeli 512MB RAM-a, 1 procesor ter 20GB prostora na SSD trdem disku.
  Na strežniku je bil izbran algoritem za mehurčno urejanje (ang. *bubble sort*). Skripto za simulacijo odjemalcev smo pognali iz študentskega naselja Rožna dolina (Ljubljana). Za dostop do interneta je bila uporabljena internetna povezana s hitrostjo 100/100 Mbps.
  Na strežnik so klienti (niti) pošiljali datoteke, ki so vsebovale velikosti 10.000, 15.000, 20.000, 25.000, 30.000, 40.000 in 50.000 števil tipa integer. Meritve smo ponovili na 1, 10, 20, 30 in 40 odjemalcih.

- **Rezultati meritve:**



Slika 8.4: Graf povprečnega časa čakanja odjemalcev v odvisnosti od velikosti datotek.

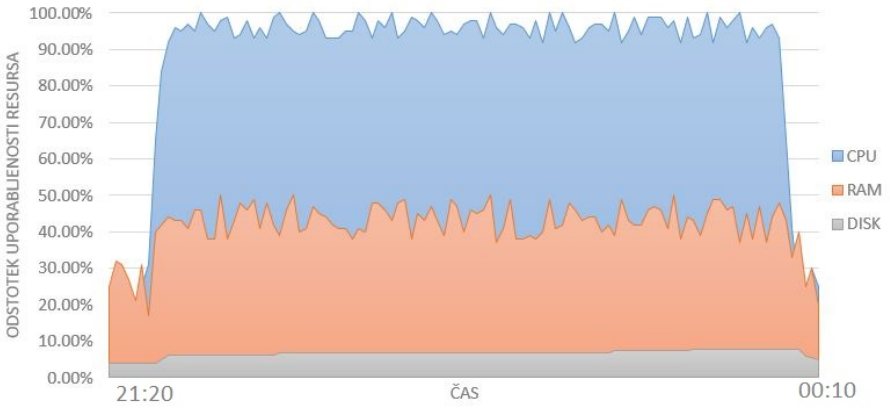
- **Komentar meritve:**
  Iz grafa na sliki 8.4 zelo razločno vidimo, da se povprečen čas čakanja posameznega odjemalca z večanjem velikosti datotek zelo hitro povečuje. Pri 40 odjemalcih in datoteki velikosti 10.000 števil tipa integer je povprečni čas čakanja približno 170 sekund. Pri datoteki veliosti 50.000 števil pa ta čas znaša že več kot 2550 sekund. To je bilo pričakovano, saj je bil za urejanje števil uporabljen algoritem za mehurčno urejanje (bubble sort), ki ima časovno zahtevnost $O(n^2)$. Rezultati tega eksperimenta potrdijo, da se tudi na oblaku potreben čas za obdelavo datotek z algoritmom *bubble sort* povečuje približno s kvadratom velikosti datotek.

8.4.3 Ozko grlo na strežniku - Eksperiment 3

- **Hipoteza:**
  Z našimi meritvami smo strežnike obremenili do najvišjih dovoljenih meja. V praksi ob maksimalni obremenitvi sistema vedno obstaja neko ozko grlo sistema, ki ne more delovati hitreje in s tem znižuje hitrosti delovanja sistema celotnega sistema. V našem primeru predpostavljamo, da je najbolj obremenjena komponenta procesor, saj je sortiranje operacija, ki zahteva relativno veliko računskih operacij.

- **Okoliščine meritve:**
  Obremenitve strežnika smo merili med Eksperimentom 2 (razdelek 8.4.2).

- **Rezultati meritve:**



Slika 8.5: Graf zasedenosti resursov med izvajanem testovs.

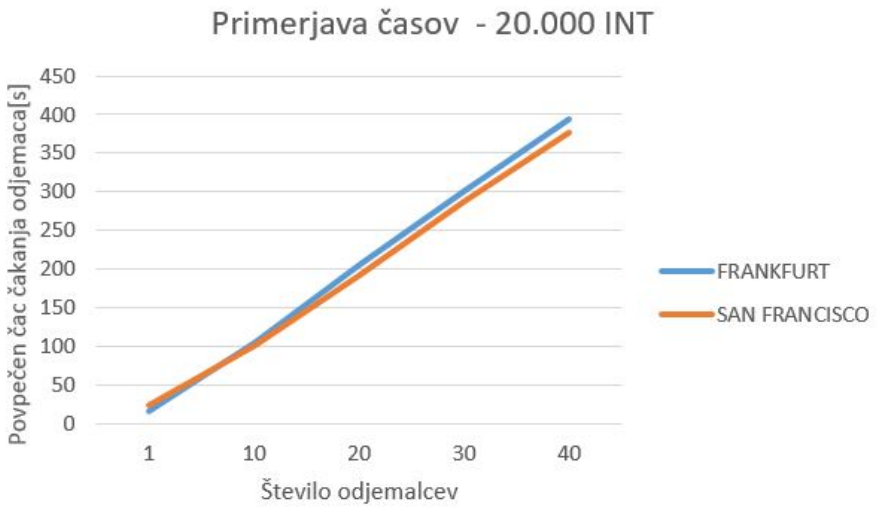
- **Komentar meritve:**
  Iz slike 8.5 lahko razločno razberemo, da je ozko grlo našega sistema procesor, saj je praktično ves čas izvajanja testov 100% obremenjen. Poraba RAM-a in kapacitete trdega diska niso nikoli ekstremno obremenjene, saj med izvajanjem nikoli ne uporabljamo več kot 50% RAM-a. Razlog za majhno porabo RAM-a je v tem, da odjemalci čakajo na odgovor strežnika pred pošiljanjem novih datotek. Tudi v primeru, ko odjemalci ne čakajo, pa nam ni uspelo doseči dovolj visoke zasedenosti, da bi zrušili sistem. Razloga za to sta omejeno število niti na odjemalcu ter omejeno število hkrati odprtih datotek na odjemalcu. Zasedenost diska pa je praktično zanemarljiva, saj so datoteke relativno majhne glede na ponujeno velikost trdega diska, datoteke pa se sproti brišejo in tako preprečimo težave s prezasedenostjo trdega diska.

8.4.4 Lokalnost - Eksperiment 4

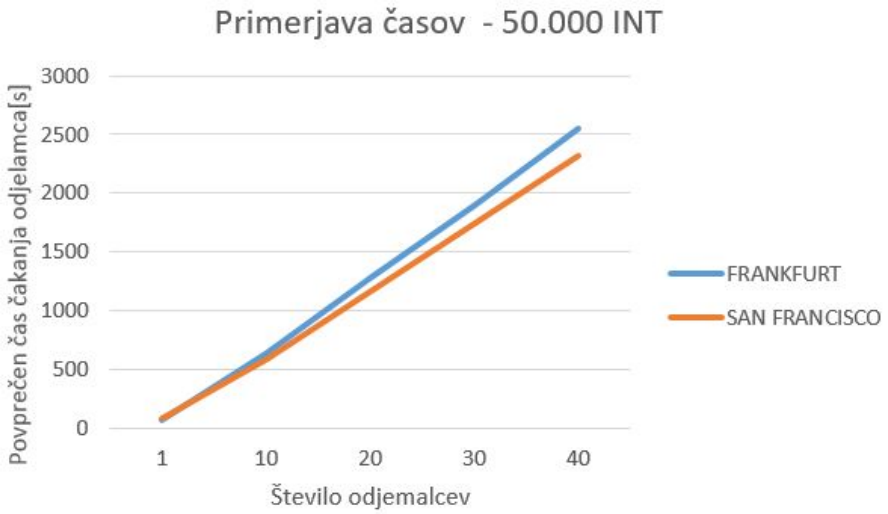
- **Hipoteza:**
  Zaradi možnosti izbire lokacije strežnika smo želeli preveriti vpliv lokacije na povprečni čas čakanja odjemalcev. Poleg strežnika v Frankfurtu smo vzpostavili tudi strežnik v San Franciscu. Predvidevamo, da bomo hitrejše odzive dobivali od strežnika v Frankfurtu, saj nam je fizično precej bližje, kot San Francisco.

- **Okoliščine meritve:**
  Strežnika imata dodeljeno enako količino resursov in nameščeno identično programsko opremo. Ta je enaka kot v testu 2 (poglavje 8.4.2) Slednje je bilo potrebno zagotoviti, da izključimo vpliv programske opreme na hitrost odziva strežnika. Testiranje smo izvedli v ponedeljek 22.5.2017 med 10:00 in 14:00 po času UTC+01:00. Zaradi časovnih pasov je bila ura v San Franciscu v času testiranja med 1:00 in 5:00 zjutraj. Teste smo ponovili z datotekami velikosti 20.000 in 50.000 celih stevil.

- **Rezultati meritve:**



Slika 8.6: Graf povprečnega časa čakanja odjemalcev v odvisnosti od lokacije na primeru datoteke z 20.000 števili.



Slika 8.7: Graf povprečnega časa cakanja odjemalcev v odvisnosti od lokacije na primeru datoteke s 50.000 števili.

- **Komentar meritve:**
  Rezultati meritev iz slik 8.6 in 8.7 niso potrdili naše hipoteze, saj se je strežnik v San Franciscu kljub večji oddaljenosti hitreje odzival in vračal urejene datoteke. Razlike so vse bolj opazne ob večji obremenitvi sistema (večje datoteke in več odjemalcev). Pri datoteki velikost 20.000 števil in enem odjemalcu lahko vidimo, da nam rezultate hitreje vrne strežnik v Frankfurtu. Rezultate si razlagamo na način, da je bil strežnik v Frankfurtu bolj obremenjen v času izvajanja testov. Pri majhnih primerih je kljub svoji obremenjenosti uspel rezultat vrniti pred strežnikom v San Franciscu. Z večanjem računske zahtevnosti problema pa je vse bolj do izraza prišla manjša zasedenost infrastrukture v San Franciscu. Drug možen razlog bi lahko bili zmogljivejši procesorji v strežnikih lociranih v San Franciscu, vendar interne informacije o zgradbi strežniških centrov žal niso javne.

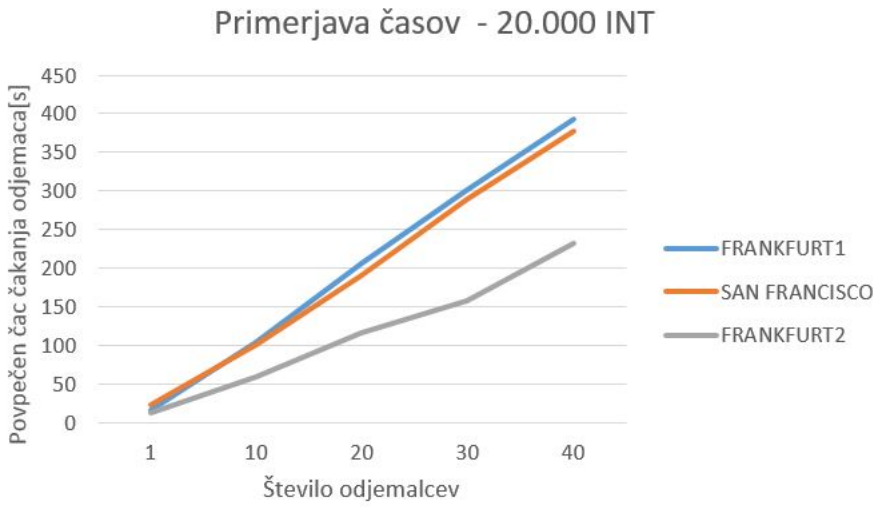
8.4.5 Različne računske moči - Eksperiment 5

- **Hipoteza:**
  Naša hipoteza, ki jo bomo preverili v tem poizkusu pravi, da se povprečni čas čakanja odjemalcev bistveno zmanjša, če je strežnik bolj zmogljiv (več procesorjev in RAM-a).

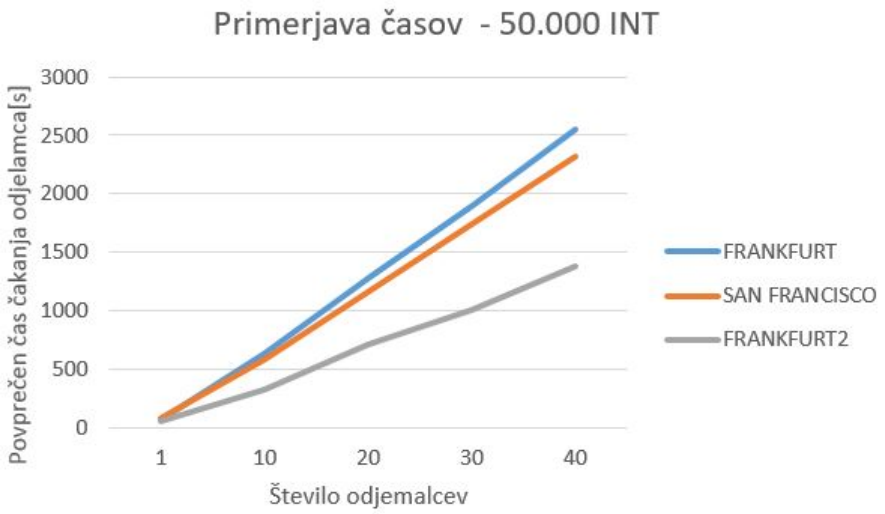
- **Okoliščine meritve:**
  Testiranje smo izvedli v torek 23.5.2017 med 00:30 in 02:30 na strežniku v Frankfurtu. Na strežniku je tekel operacijski sistem Linux Ubuntu 16.04. Na voljo smo imeli 2 GB RAM-a ter 2 procesorja. Na strežniku je bil izbran algoritem za mehurčno urejanje (ang. *bubble sort*).
  Skripto za simulacijo odjemalcev smo pognali iz študentskega naselja Rožna dolina (Ljubljana). Za dostop do interneta je bila uporabljena internetna povezana s hitrostjo 100/100Mbps.
  Na strežnik so klienti (niti) pošiljali datoteke velikosti 20.000 in 50.000 števil tipa integer in merili čase pri 1, 10, 20, 30 in 40 odjemalcih.

- **Rezultati meritve:**

  V grafih na slikah 8.8 in 8.10 so vizualizani povprečni časi odziva treh različnih streznikov. Strežnik Frankfurt1 in San Francisco imata identično konfiguracijo, kot je opisana v eksperimentu 4 (razdelek 8.4.4). Strežnik Frankfurt2 pa ima na voljo 2 procesorja ter 2 GB RAM-a.



Slika 8.8: Graf povprečnega časa čakanja odjemalcev treh različnih strežnikov na primeru datotek z 20.000 števili.



Slika 8.9: Graf povprečnega časa čakanja odjemalcev treh različnih strežnikov na primeru datotek s 50.000 števili.

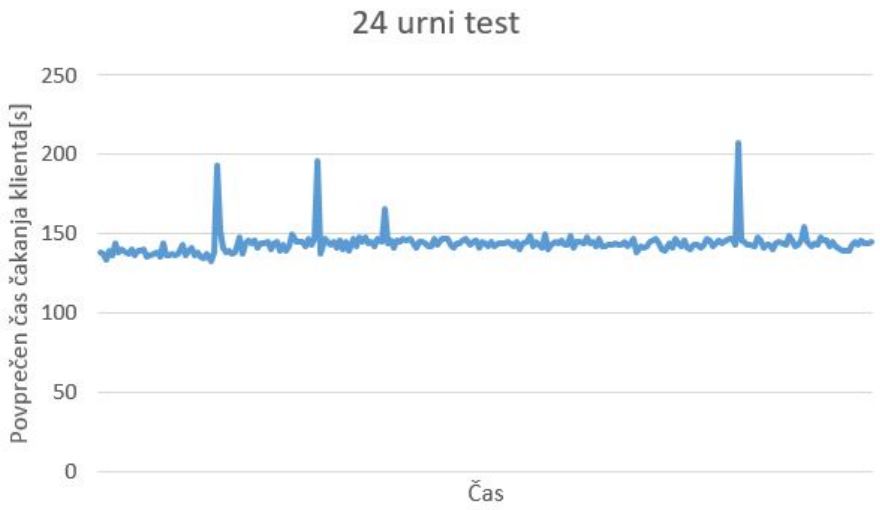
- **Komentar meritve:**
  Meritve potrdijo našo predpostavko, da višja zmogljivost strežnikov pripomore k bistveno nižjim povprečnim časom potrebnim za odziv.

8.4.6 24 urni test - Eksperiment 6

- **Hipoteza:**
  Zaradi ugotavljanja zasedenosti omrežja ter strežnikov smo izvajali še 24 ur trajajoči test. Naša predpostavka je, da bodo časi odzivov višji v dnevnih časovnih razmerah, tj. čez dan kot pa ponoči.

- **Okoliščine meritve:**
  Testiranje smo izvedli v petek 26. in v soboto 27.5.2017. Test smo začeli izvajatvi v petek ob 13:30 in smo ga izvajali 24 ur na strežniku v Frankfurtu. Na strežniku je tekel operacijski sistem Linux Ubuntu 16.04. Na voljo smo imeli 2GB RAM-a, 2 procesorja ter 40 GB prostora na SSD trdem disku. Na strežniku je bil izbran algoritem za mehurčno urejanje (ang. *bubble sort*).
  Skripto za simulacijo odjemalcev smo pognali v Trzinu. Za dostop je bila uporabljena internetna ponudba Telekom Slovenije s hitrostjo 20/10Mbps. Na strežnik so klienti (niti) pošiljali datoteke velikosti 20.000 celih števil za sortiranje v intervalih po 300 sekund. Slednje je počelo 20 odjemalcev hkrati. Merili smo povprečen čas čakanja odjemalcev in kako se ta spreminja skozi celoten dan.

- **Rezultati meritve:**



Slika 8.10: Graf povprečnega časa čakanja odjemalcev v 24 urah.

- **Komentar meritve:**
  Z meritvijo smo ugotovili, da je povprečen čas čakanja odjemalcev preko

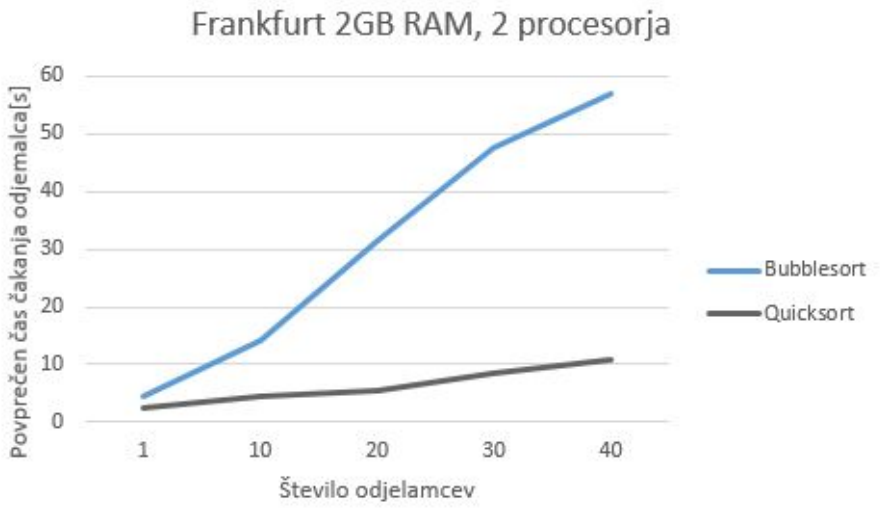
celega dneva praktično enak. To si razlagamo tako, da strežniki nikoli niso tako obremenjeni, da bi to vplivalo na odzivne čase. Čas prenašanja datotek pa zaradi razmeroma hitrih internetnih povezav predstavlja manjši delež skupnega časa. Iz povprečja bolj očitno odstopajo le štiri meritve, ki so trajale občutno dalj časa. Te meritve pa nam ne povedo veliko uporabnega, saj so razporejene dokaj naključno čez celoten dan. Naša hipoteza, ki predpostavlja, da bodo časi obdelave sredi dneva občutno višji od tistih ponoči, je bila napačna.

8.4.7 Izbira algoritma - Eksperiment 7

- **Hipoteza:**
  Na strani strežnika smo implementirali dva različna algoritma za urejanje števil opisana v razdelku 8.2.4. V tem eksperimentu bomo iste naključne datoteke pošiljali na strežnik ter merili čase do prejema datotek pri obeh algoritmih. Naša hipoteza je, da bo strežnik na katerem teče algoritem *quicksort* (Listing 8.1) z nalogo opravil veliko hitreje, kot strežnik z izbranim algoritmom *bubblesort* (Listing 8.2).

- **Okoliščine meritve:**
  Testiranje smo izvedli v sredo 31.5.2017 med 15:00 in 16:00 na strežniku v Frankfurtu. Na strežniku je tekel operacijski sistem Linux Ubuntu 16.04. Na voljo smo imeli 2 GB RAM-a ter 2 procesorja. Skripto za simulacijo odjemalcev smo pognali iz študentskega naselja Rožna dolina (Ljubljana). Za dostop do interneta je bila uporabljena internetna povezana s hitrostjo 100/100Mbps.
  Na strežnik so klienti (niti) pošiljali datoteke velikosti 20.000 števil tipa integer in merili čase pri 1, 10, 20, 30 in 40 odjemalcih.

- **Rezultati meritve:**
  Na sliki 8.11 lahko vidimo gafično predstavljene rezultate meritve.



Slika 8.11: Graf povprečnega časa čakanja odjemalcev pri različnih algoritmih.

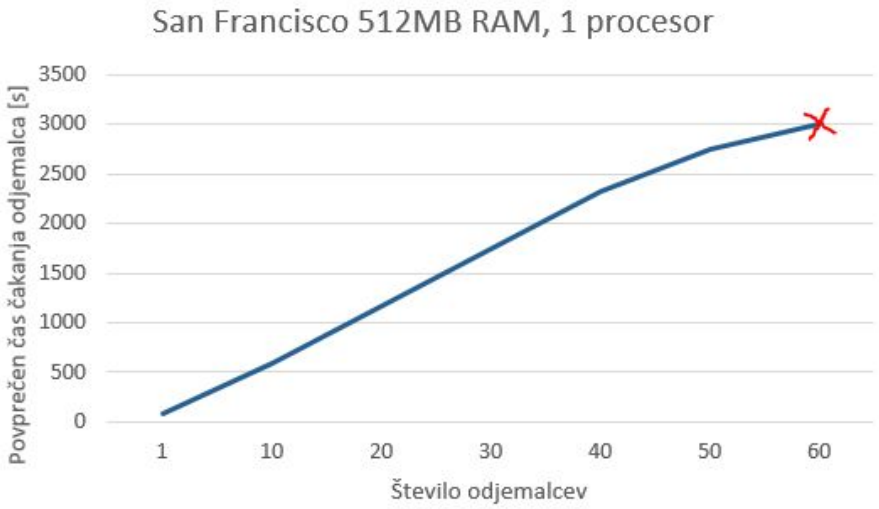
- **Komentar meritve:**
  Rezultati vidni na sliki 8.11 potrjujejo našo hipotezo. Čas sortiranja na strežniku na naključnih datotekah je precej manjši ob uporabi algoritma *quicksort*.

8.4.8 Zlom sistema - Eksperiment 8

- **Hipoteza:**
  Za konec smo želeli tudi zlomit sistem oz. ga prisiliti v to, da nam ne vrača več pravilnih rezultatov. Naša hipoteza je, da bomo prej ali slej prišli do točke, ko bo strežnik zaradi predolgega časa trajanja, potrebnega za odgovor, povezavo zaprl.

- **Okoliščine meritve:**
  Testiranje smo izvedli v četrtek 1.6.2017 med 19:00 in 20:30 na strežniku v San Franciscu. Na strežniku je tekel operacijski sistem Linux Ubuntu 16.04. Na voljo smo imeli 512MB RAM-a ter 1 procesor. Na strežniku je bil izbran algoritem za mehurčno urejanje (ang. *bubble sort*).
  Skripto za simulacijo odjemalcev smo pognali iz študentskega naselja Rožna dolina (Ljubljana). Za dostop do interneta je bila uporabljena internetna povezana s hitrostjo 100/100Mbps.
  Na strežnik so klienti (niti) pošiljali datoteke velikosti 50.000 števil tipa integer in merili čase pri 1, 10, 20, 30, 40, 50, 60 in 80 odjemalcih.

- **Rezultati meritve:**



Slika 8.12: Graf povprečnega časa čakanja odjemalcev pri uspešnem poizkusu zloma sistema.

- **Komentar meritve:**
  Z eksperimentom, katerega rezultati so prikazani na sliki 8.12, nam je uspelo uspešno doseči zlom sistema, saj je strežnik po približno 3000 s čakanja na odgovor povezavo prekinil. Podrobnejše sporočilo o napaki je na vojo v listingu 8.3. Ker nismo vedeli na kakšen način in kdaj bo prišlo do zloma strežnika, je bilo potrebno veliko poizkusov, da smo dosegli željen rezultat.

Listing 8.3: Napaka ob zlomu sistema.

```
Traceback (most recent call last): File
    "C:\Anaconda3\lib\site-packages\requests\adapters.py", line
    324, in send timeout=timeout
  File "C:\Anaconda3\lib\site-packages\requests\packages\urllib3\
      connectionpool.py", line 528, in urlopen raise
      MaxRetryError(self, url, e)
requests.packages.urllib3.exceptions.MaxRetryError:
    HTTPConnectionPool(host='139.59.132.145', port=8080): Max
    retries exceeded with url: /nalozi (Caused by <class
    'TimeoutError'>: [WinError 10060] A connection attempt failed
    because the connected party did not properly respond after a
    period of time, or established connection failed because
    connected host has failed to respond)
```

8.5 Zaključek

Naš cilj je bil preveriti zmogljivost oblačnih storitev za izvajanje računsko zahtevnih operacij. Uporabili smo strežniško infrastrukturo podjetja DigitalOcean in na njej izvajali operacijo urejanja števil, saj ta vsebuje veliko operacij. Izvajali smo različne teste, ter tako preverjali vpliv določenih spremenljivk na zmogljivost našega sistema.
S testi smo hoteli ugotoviti vplive števila odjemalcev, velikosti datotek, lokacije strežnika, količine razpoložljivih resursov na strežniku, ure v dnevu ter izbire algoritma na hitrost odziva strežnika. Poleg tega smo izvedli tudi test s katerim smo povzročili zlom sistema. Merili smo predvsem čase potrebne za pošiljanje in sortiranje datotek ter zasedenost resursov na strežniku.
Kot glavno ozko grlo strežnika se je pričakovano izkazal CPU, saj mora pri sortiranju opraviti zelo veliko operacij primerjanja. Maksimalno zasedenost CPU-ja laho hitro dosežemo z večanjem števila odjemalcev in velikosti datotek.
Za metrike smo se odločili na podlagi članka [87]. V veliko pomoč pa so nam bile tudi ugotovitve iz člankov [88], [89] in [90]. Nekatere podobne meritve smo našli v članku [91], kjer so ravno tako kot mi merili zmogljivost programov v oblaku. Na podlagi eksperimentov smo ugotovili, da je izbira algoritma zelo pomembna in bistveno vpliva na čas obdelave. Prav tako je ključna obremenjenost strežnika s številom klientov. Kljub vsemu, pa strežnik s tako nizkimi zmogljivostmi ne pride v poštev za praktičo uporabo reševanja računskih problemov namesto končnih delovnih točk (osebnih računalnikov). Ob tem pa je potrebno poudariti, da na trgu obstajajo tudi precej zmogljivejše rešitve, ki omogočajo bistveno večjo računsko moč. Vedno hitrejši strežniki in povečevanje povprečne hitrosti internetne povezave, pa bo slej ko prej pripeljalo do točke, ko bomo iz končnih točk računsko zahtevne probleme v obdelavo pošiljali na strežnik. Naše naprave (končne točke), pa bodo skrbele le za interakcijo s strežnikom in prikaz rezultatov.

Poglavje 9

Analiza zmogljivosti oblačnih storitev različnih ponudnikov

Blaž Dolenc, Matej Šnuderl,
Jakob Merljak

9.1 Opis problema

V tem poglavju želimo preizkusiti različne ponudnike oblačnih storitev iz vidika uporabe modernih spletnih aplikacij - torej takšnih, pri katerih se generirajo velike količine podatkov in prometa. Posledično se želimo osredotočiti na testiranje zmogljivosti shrambe, diska in procesorske moči. Teste bomo skušali čimbolj približati realni uporabi, torej bomo pozorni na parametre, ki vplivajo na uporabniško izkušnjo končnega uporabnika aplikacije, ki gostuje pri ponudniku oblačne storitve. S tem bomo skušali olajšati izbiro najboljšega ponudnika.

9.2 Izbira tehnologij

Naša izbira ponudnikov oblačnih storitev, katere bomo testirali v seminarski nalogi, bo temeljila na njihovi razširjenosti, pa tudi na tem, kakšne brezplačne ali nizkocenovne možnosti za gostovanje spletnih aplikacij ponujajo. Skušali bomo izkoristiti študentske ugodnosti in zagotoviti virtualne strežnike, ki bodo čimbolj podobni produkcijskim zahtevam. Po pregledu ponudbe smo se odločili za sledeče ponudnike: Amazon Web Services [92], Microsoft Azure [93] in Digital Ocean [94]. Našteti ponudniki omogočajo pridobitev brezplačnih virtualnih strežnikov na katerih bomo izvajali meritve, poleg tega pa je njihova baza uporabnikov zelo široka. Na virtualnem strežniku vsakega izmed ponudnikov

bomo izvedli nabor testov. Na vseh platformah smo kreirali virtualni strežnik z operacijskim sistemom Ubuntu 16.04.

Za izvajanje meritev smo izbrali več odptokodnih orodij za testiranje zmogljivosti strežnikov.

9.3 Specifikacije izbranih virtualnih strežnikov

Tabela 9.1 kaže, kakšne pakete zmogljivosti smo dobili pri posameznem ponudniku. Odločili smo se za cenovno primerljive strežnike, saj je cena pogosto odločilna, ko govorimo o izbiri gostovanja za aplikacije. Iz tabele prav tako izhaja, da večjih razlik vsaj na papirju med izbranimi ponudniki ni. Vsi uporabljajo procesorje iz družine Intel Xeon, prav tako sta na vseh virtualnih strežnikih na voljo 2 jedri. Razlika je v količini pomnilnika in diska, kar pa glede na to, da primerjamo zmogljivost predvsem iz stališča zmogljivosti ni ključnega pomena.

Tabela 9.1: Specifikacije virtualnih strežnikov.

Ponudnik	Processor	Pomnilnik	Trdi disk	Mesečna cena
AWS	Xeon E5-2680, 2.8GHz	3.75GB	SSD, 16GB	20EUR
Azure	Xeon E5-2660, 2.2GHZ	7GB	SSD, 60GB	25EUR
Digital Ocean	Xeon E5-2650, 1.8GHz	2GB	SSD, 40GB	20EUR

9.4 Definicija bremena storitve

Oblačne storitve bomo med seboj primerjali po procesorski moči, hitrosti trajnega pomnilnika in delovnega pomnilnika. Te karakteristike najpomembneje vplivajo na zmogljivost končnega sistema. Za meritev teh zmogljivosti bomo uporabljali namenske programe, ki omogočajo simuliranje operacij in transakcij na ciljnem sistemu in posledično merjenje in primerjavo zmogljivosti sistemov.

Da bo končni rezultat čimbolj reprezentativen in ponovljiv smo se odločili za več iteracij testov, poleg tega pa smo se odločili za bremena, ki variirajo po tipu. Cilj tega je, da z večimi različnimi sintetičnimi testi testiramo odziv sistema, končni rezultati pa nam dajo predstavo, kako bi se posamezen ponudnik obnesel pod realnim bremenom.

9.5 Definicija metrik in orodij za meritve

V sledečih razdelkih so zbrane metrike in orodja, ki smo jih uporabili za testiranje posameznih področij.

9.5.1 Procesor

Procesor je ena ključnih komponent modernih računalniških sistemov. Prav zato smo mu namenili posebno pozornost, saj v dobi velikih količin podatkov (Big data) in podatkovnega rudarjenja aplikacije, ki tečejo na strežnikih postajajo vse bolj procesorsko zahtevne. Posledično je pri izbiri ustreznega oblačnega ponudnika zmogljivosti procesorja nameniti posebno skrb. Poleg zmogljivosti je seveda pomembna tudi cena, saj gostovanja z zelo zmogljivimi procesorji lahko stanejo tudi večkratnik osnovne ponudbe. Tip bremena, ki teče na strežniku pomembno vpliva na izbiro ustrezne instance pri oblačnem ponudniku. Metrike, ki so pomembne pri procesorju, so takt procesorja (število ciklov, ki jih lahko izvede), število jeder itd. Spodaj našteti testi testirajo različne aspekte teh dveh metrik. Vsi testi za procesor so bili izvedeni direktno na strežniku preko ukazne lupine. Virtualni strežniki niso imeli nameščenega grafičnega okolja.

Iskanje praštevil

Iskanje praštevil je klasičen test za testiranje zmogljivosti procesorjev. Algoritem, ki ga uporablja test, uporablja deljenje in primerjave števil. Posledično test preverja manjši set funkcionalnosti procesorja, a po drugi strani zelo pomembne, kar je razlog, da je to eden izmed testov vključenih v izbor.

Ker smo želeli poleg zmogljivosti za eno nit preveriti tudi, kako se ponudniki obnesejo pri več nitnem procesiranju, smo test pognali še z 4 nitmi.

Orodje sysbench [95] ponuja test, ki izvede iskanje praštevil, pri čemer podamo zgornjo mejo, do katere testiramo. Program izpiše čas, porabljen za iskanje prvih n praštevil. Izpis 9.1 prikazuje primer testa procesorja na platformi Azure, na sistemu Ubuntu Linux. V podanem primeru se računa prvih 20 000 praštevil:

Listing 9.1: Primer testiranja procesorja.

```
zzrs@ZZRS:~$ sysbench --test=cpu --cpu-max-prime=20000 run
sysbench 0.4.12: multi-threaded system evaluation benchmark

Running the test with following options:
Number of threads: 1

Doing CPU performance benchmark

Threads started!
Done.

Maximum prime number checked in CPU test: 20000


Test execution summary:
    total time:                          29.6635s
    total number of events:              10000
    total time taken by event execution: 29.6616
```

```
    per-request statistics:
         min:                                  2.55ms
         avg:                                  2.97ms
         max:                                 56.83ms
         approx. 95 percentile:                3.33ms

Threads fairness:
    events (avg/stddev):           10000.0000/0.00
    execution time (avg/stddev): 29.6616/0.00
```

Glavna metrika je čas izvajanja, kar bo dober indikator med razlikami procesorske moči, ki jo ponujajo različni ponudniki.

Stiskanje datotek z programom Gzip

Precej pogosta in procesorsko zahtevna operacija v računalništvu je stiskanje datotek, pri čemer se uporabljajo različni algoritmi. Program gzip temelji na algoritmu DEFLATE [96], ki je kombinacija LZ77 in Huffmanovega kodiranja. Med kompresiranjem se izvede množica primerjanj in iskanj blokov, ponavljajoči deli pa se nato zakodirajo.

Orodje Phoronix test suite [97] med drugim ponuja tudi test stiskanja z uporabo programa gzip 9.2. Test se izvede večkrat, uporabili pa smo povprečno vrednost. Izvede se stiskanje datotek in meri porabljen čas.

Listing 9.2: Primer testiranja procesorja z programom gzip.

```
Gzip Compression:
    pts/compress-gzip-1.1.0
    Test 1 of 1
    Estimated Trial Run Count: 3
    Estimated Time To Completion: 3 Minutes
        Running Pre-Test Script @ 18:53:42
        Started Run 1 @ 18:54:11
        Started Run 2 @ 18:54:44
        Started Run 3 @ 18:55:10 [Std. Dev: 0.48%]
        Running Post-Test Script @ 18:55:34

    Test Results:
        23.839015960693
        23.654896974564
        23.861775875092

    Average: 23.79 Seconds
```

Razbijanje gesel

Zadnji test, ki smo ga uporabili za testiranje procesorja in ki se pogosto izvaja na temnih straneh interneta, je razbijanje zgoščenih vrednosti gesel. Za hra-

njenje gesel se namreč uporablja enosmerne funkcije, s čimer zagotavljamo, da napadalec kljub dostopu do baze, ki vsebuje podatke o uporabnikih, ne pridobi tudi njihovih gesel. Če napadalec torej želi uporabiti pridobljene zgoščene vrednosti, jih mora razbiti - uganiti, kateri niz znakov proizvede enako zgoščeno vrednost.

Orodje phoronix nam ponuja izbiro več zgoščevalnih funkcij. Odločili smo se za DES, ki je pogosto uporabljena za hranjenje gesel. Z pomočjo slovarja nato orodje simulira razbijanje gesel. Tudi ta test je procesorsko zelo zahteven, kar predstavlja dobro breme za testiranje.

9.5.2 Delovni pomnilnik

Delovni pomnilnik (RAM) je eden najpomembnejših elementov računalniškega sistema. Njegova naloga je shranjevanje pogosto uporabljenih ukazov do katerih nam omogoča zelo hiter dostop z namenom povečanja splošne hitrosti sistema. V primerjavi z drugimi oblikami hrambe podatkov je relativno hiter, vendar ni persistenten.

Pisanje in branje iz pomnilnika

Pri testiranju delovnega pomnilnika bremenska aplikacija alocira medpomnilnik in nato izvaja write in read ukaze v velikosti kazalca (32 ali 64 bitov), v vsaki izvedbi dokler ni celoten alociran medpomnilnik (–memory-block-size) označen kot prebran/prepisan. Vse skupaj se ponavlja dokler ni dosežena kvota (–memory-total-size). Pseudokoda je prikazana na izpisu 9.3.

Listing 9.3: Okvirna koda za test delovnega pomnilnika.

```
total_bytes = 0;
repeat until total_bytes >= memory-total-size:
  thread_mutex_lock()
  total_bytes += memory-block-size
  thread_mutex_unlock()

  (timer start)
  pointer -> buffer;
  while pointer <-> end-of(buffer)
    beri/pisi vrednost na lokacijo pointera
    pointer++
  (timer end)
```

Uporabnik lahko navede tudi več niti izvedbe (–num-thread) in tipe ukazov (branje ali pisanje, zaporedno ali naključno). Pri naših testih smo uporabili parametre uporabljene v izpisu 9.4, saj so se nam zdeli podobni realnim zahtevam po pomnilniku v večjih in požrešnih sistemih.

Metrika

Metrika, ki nas zanima pri testu delovnega pomnilnika, je število operacij na sekundo in pretok (količina prenešenih podatkov) na sekundo. Tako dobimo hitrost operacij na sekundo, ki jih je pomnilnik zmožen izvesti. To lahko zelo vpliva na splošno hitrost sistema, saj je pomnilnik v programih dostopan neprestano. Test je lahko reproducibilen in se lahko izvede kot je prikazano na izpisu 9.4.

Listing 9.4: Primer testiranja pomnilnika.

```
sysbench memory --memory-block-size=1K --memory-total-size=100G
    --memory-access-mode=rnd --memory-oper=read run
sysbench 1.0.6 (using bundled LuaJIT 2.1.0-beta2)

Running the test with following options:
Number of threads: 1
Initializing random number generator from current time


Running memory speed test with the following options:
  block size: 1KiB
  total size: 102400MiB
  operation: read
  scope: global

Initializing worker threads...

Threads started!

Total operations: 10370838 (1036881.13 per second)

10127.77 MiB transferred (1012.58 MiB/sec)


General statistics:
    total time:                          10.0001s
    total number of events:              10370838

Latency (ms):
         min:                                    0.00
         avg:                                    0.00
         max:                                    0.09
         95th percentile:                        0.00
         sum:                                 6921.53

Threads fairness:
    events (avg/stddev):           10370838.0000/0.00
    execution time (avg/stddev): 6.9215/0.00
```

9.5.3 Trdi disk

Trdi disk (obstojni pomnilnik) je še eden izmed nepogrešljivih sestavnih delov v večini računalniških sistemov. Je zelo pomemben dejavnik zmogljivosti sistemov. Izkaže se tudi, da je v veliko sistemih prav trdi disk ozko grlo.

Metrike

Zmogljivost diska je odvisna od treh njegovih karakteristik. Za vsako obstajajo ustrezne metrike:

- Število vhodno/izhodnih operacij v časovni enoti - večje kot je število, večja je zmogljivost diska. Potrebno se je zavedati, da je to število zelo odvisno od tega, kakšne so te operacije in kako velike podatke se prenaša - če se npr. prenašajo veliki kosi podatkov (npr. velike datoteke), se bo lahko v neki časovni enoti izvedlo manj operacij, kot če bi v isti časovni enoti prenašali manjše kose podatkov. Metrika, ki se uporablja je IOPS (Input/Output Operations per Second) - število vhodno/izhodnih operacij na sekundo.
- Čas, potreben za začetek operacije oz. latenca (ang. latency) - kako hitro se lahko disk odzove in začne prenos. Krajši kot je čas, večja je zmogljivost diska. Meri se v sekundah (lahko tudi manjših enotah, odvisno od programa), čeprav so latence po navadi veliko manjše od sekunde.
- Hitrost prenašanja podatkov - koliko podatkov se lahko prenese na disk v neki časovni enoti. Višja vrednost pomeni višjo zmogljivost. Meri se v bajtih (kilobajtih, megabajtih) na sekundo.

Breme in orodja za meritev

Večina računalniških sistemov ne ponuja neposrednega dostopa in upravljanja z diskom. Večina operacijskih sistemov, tudi v našem primeru, ponuja dostop do diska preko datotečnega sistema, torej deloma v režiji operacijskega sistema. Ta zaradi boljše izrabe omejenih virov uporablja medpomnilnik in različne strategije prenašanja podatkov na diske. To pomeni, da se zmogljivosti enakih diskov v sistemih z različnimi operacijskimi sistemi in seveda bremeni, lahko razlikujejo. Na večini platform programi za merjenje zmogljvosti diska pišejo in berejo datoteke, kar se na koncu izkaže za zelo univerzalno breme, ki daje primerljive rezultate.

Zaradi bločne organiziranosti diskov se pojavijo tudi razlike med naključnim ter zaporednim branjem/pisanjem. Za celovitost preizkusov je torej potrebno testirati tako zaporedno kot naključno branje in pisanje.

Za meritev IOPS in hitrosti prenosa podatkov bomo uporabili program *fio* [98]. Program zna simulirati najrazličnejše vrste obremenitve diska, ki se jih da poljubno nastavljati, npr. razmerje vhodnih in izhodnih operacij. Za merjenje latence bomo uporabili program *ioping* [99].

9.6 Rezultati

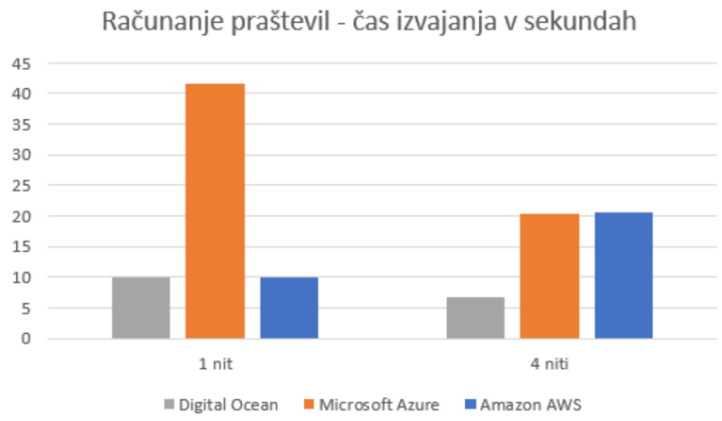
Pri testiranju smo se osredotočili na ponovljivost rezultatov in večkratno izvajanje, da bi bili dobljeni podatki v največji meri reprezentativni. Za posamezna področja testiranja je tako najprej opisan postopek poganjanja bremen, zbrani rezultati, ter komentar. Na koncu podamo še komentar rezultatov glede na to, kateri ponudnik se je v celoti gledano izkazal najbolje.

9.6.1 Procesor

Rezultati v tabeli 9.2 prikazujejo koliko časa je posamezen ponudnik porabil za iskanje 20 000 praštevil. Čas je podan v sekundah, manjša vrednost pomeni boljši rezultat. Teste smo izvedli 5x in izračunali povprečje. Poleg tega smo jih izvajali ob različnih časih dneva, a nismo zaznali razlik, ki bi bile dovolj signifikantne. Iz rezultatov lahko sklepamo, da je v primerjavi z AWS in Digital Ocean platforma Azure procesorsko kar se golega računanja tiče nekoliko manj zmogljiva, kar je razvidno tudi iz grafa za 1 nit na sliki 9.1.

Tabela 9.2: Računanje praštevil.

Ponudnik	Test	1 nit [s]	4 niti [s]
AWS	Max prime	10.01	20.64
Azure	Max prime	41.63	20.43
Digital Ocean	Max prime	9.98	6.68



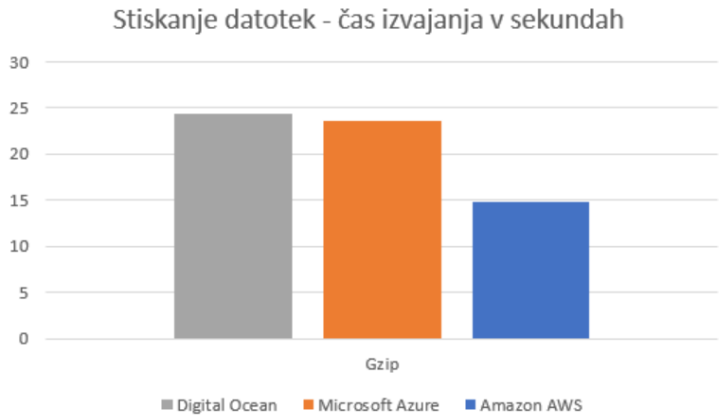
Slika 9.1: Test CPU - praštevila.

Test stiskanja datotek smo izvedli pod enakimi pogoji kot test iskanja praštevil. Izkaže se, da kljub temu, da je platforma Digital Ocean superiorna pri

metriki računanja praštevil v primerjavi z Azurom, pri tem testu razlike praktično ni. To samo potrdi tezo, da je potrebnih več različnih sintetičnih testov, če želimo, da so rezultati testiranja reprezentativni in uporabni.

Tabela 9.3: Stiskanje datotek.

Ponudnik	Test	Čas izvajanja [s]
AWS	gzip	14.77
Azure	gzip	23.65
Digital Ocean	gzip	24.32



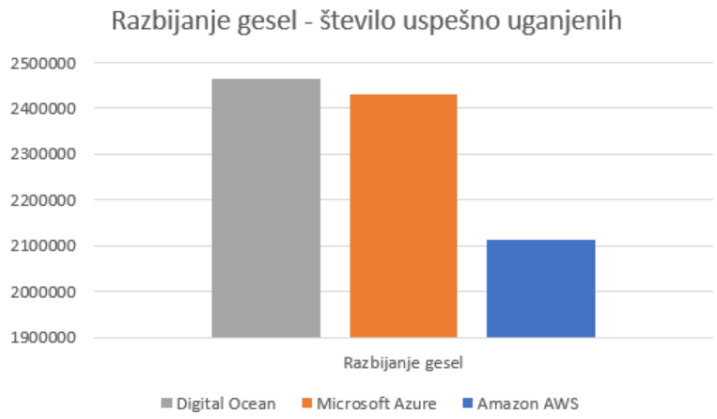
Slika 9.2: Test CPU - stiskanje datotek.

Rezultati razbijanja gesel v tabeli 9.4 prikazujejo, koliko gesel je bilo uspešno razbitih na posamezni platformi. Test smo izvedli trikrat in vzeli povprečno vrednost. Večja vrednost pomeni boljši rezultat. Iz rezultatov v tabeli 9.4 lahko sklepamo, da je ponovno vodilna platforma Digital Ocean, kljub temu da so razlike majhne (slika 9.3).

Tabela 9.4: Razbijanje gesel.

Ponudnik	Test	Število razbitih gesel
AWS	Razbijanje gesel	2114333
Azure	Razbijanje gesel	2428667
Digital Ocean	Razbijanje gesel	2463833

Povzetek vseh rezultatov testiranja procesorja nam pokaže, da so razlike pri procesorski moči, ki nam je na voljo v osnovi majhne, edina večja razlika se



Slika 9.3: Test CPU - Razbijanje gesel.

je izkazala pri testu računanja praštevil, kjer je poudarek na goli aritmetiki. Ker smo teste izvajali ob različnih urah (posledično tudi ob različni obremenitvi oblačne storitve na splošno) rezultati pa so bili konsistentni lahko sklepamo, da vsaj za manjše instance virtualnih strežnikov, kot smo jih uporabljali mi dobimo dedicirano procesorsko moč.

Zanimiva je tudi primerjava ponudnika AWS, saj se je pri gzip testu odrezal nadpovprečno, medtem, ko rezultat za razbijanje gesel in računanje praštevil te prednosti ne zaznaza.

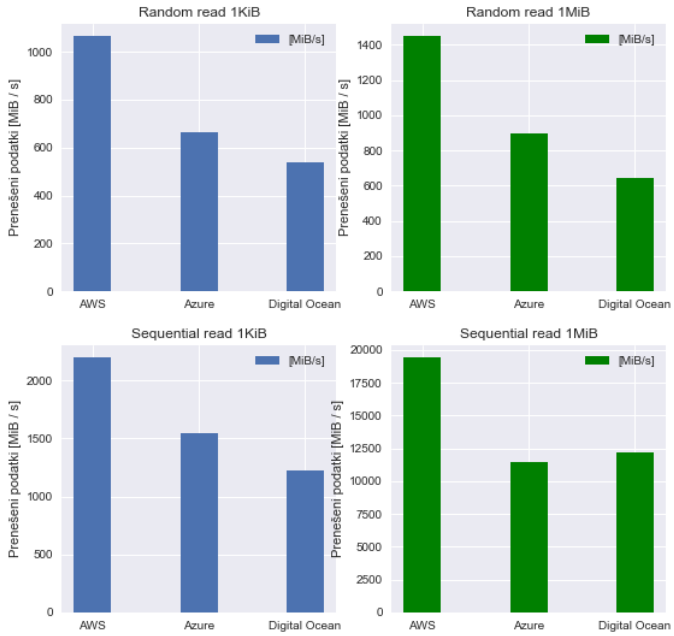
Ravno tako lahko potegnemo zaključek, da pri večini bremen razlike v zmogljivosti procesorja ne predstavljajo bistvene prednosti za posameznega ponudnika. To pomeni, da se lahko pri izbiri osredotočimo na druge dejavnike, kot je cena in težavnost konfiguracije, ter podpora.

9.6.2 Delovni pomnilnik

Po analizi podatkov pridobljenih z izvedenimi bremenitvenimi testi smo ugotovili da se na področju delovnega pomnilnika upoštevajoč metriko ki smo jo opisali najboljše odreže AWS. Kot je razvidno iz tabele 9.5 in slike 9.4, AWS pri branju ukazov iz pomnilnika v povprečju izvede približno 1.6x več operacij kot Azure in 2.0x več operacij kot Digital Ocean. Sorazmerno s tem se poveča tudi količina prenešenih podatkov. Ujemajoče rezultate opazimo tudi pri pisanju v pomnilnik v tabeli 9.6 in sliki 9.5.

Tabela 9.5: Rezultati bremenitvenih testov branja iz pomnilnika.

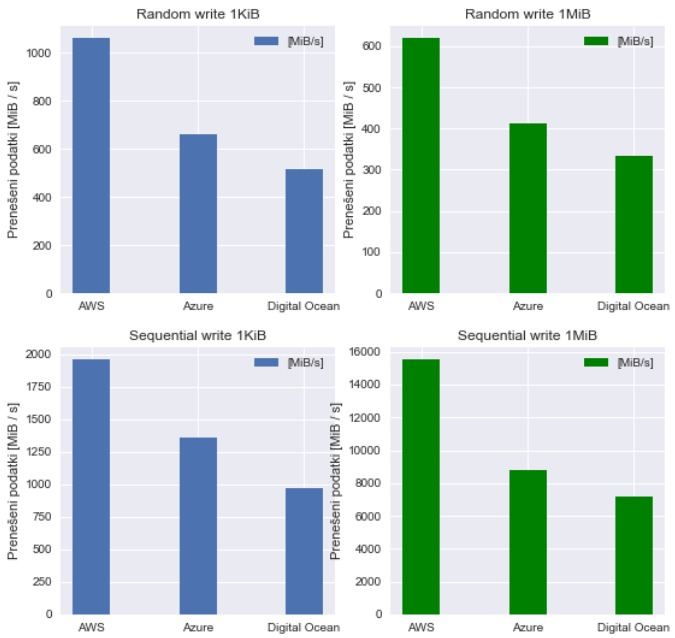
Ponudnik	Zaporedje	Vel. bloka	Prenešeno [MiB/s]	Operacije [/s]	Čas izvajanja
AWS	naklj.	1KiB	1063.85	1089387.31	10.00s
Azure	naklj.	1KiB	663.57	679498.62	10.00s
Digital Ocean	naklj.	1KiB	535.79	548653.50	10.00s
AWS	zap.	1KiB	2197.02	2249753.04	10.00s
Azure	zap.	1KiB	1547.44	1584578.26	10.00s
Digital Ocean	zap.	1KiB	1220.04	1249319.93	10.00s
AWS	naklj	1MiB	1448.83	1448.83	10.00s
Azure	naklj.	1MiB	897.21	897.21	10.00s
Digital Ocean	naklj.	1MiB	646.84	646.84	10.00s
AWS	zap	1MiB	19409.32	19409.32	5.27s
Azure	zap.	1MiB	11480.77	11480.77	8.92s
Digital Ocean	zap.	1MiB	12232.52	12232.52	8.37s



Slika 9.4: Grafi branja iz pomnilnika.

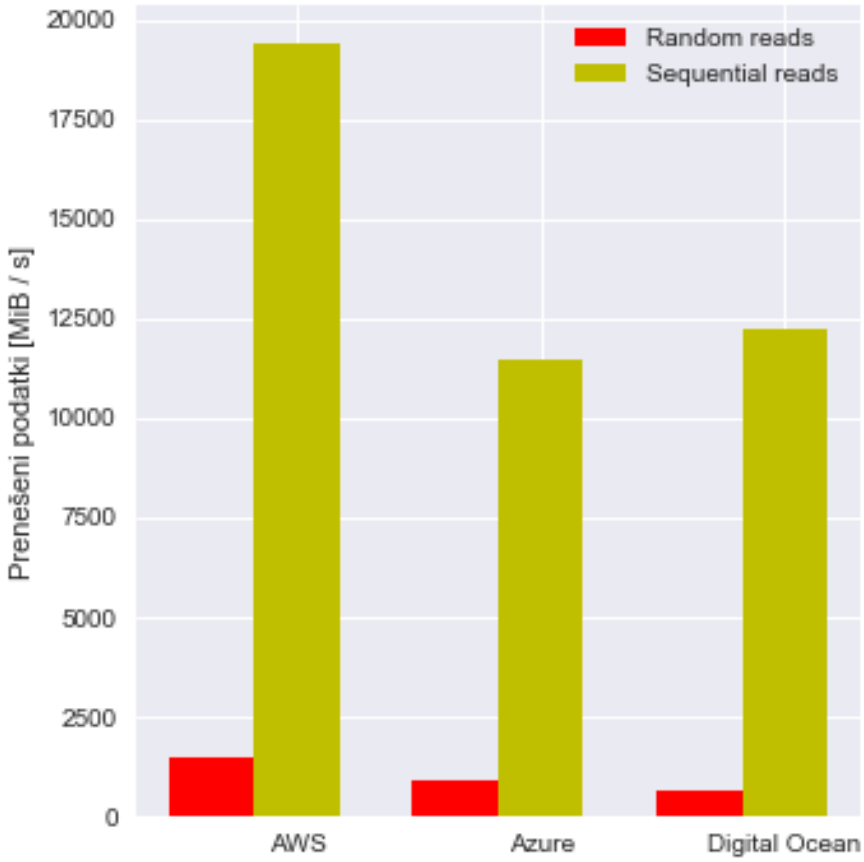
Tabela 9.6: Rezultati bremenitvenih testov pisanja v pomnilnik.

Ponudnik	Zaporedje	Vel. bloka	Prenešeno [MiB/s]	Operacije [/s]	Čas izvajanja
AWS	naklj.	1KiB	1059.79	1085220.89	10.00s
Azure	naklj.	1KiB	658.80	674606.28	10.00s
Digital Ocean	naklj.	1KiB	513.92	526252.88	10.00s
AWS	zap.	1KiB	1957.00	2003966.84	10.00s
Azure	zap.	1KiB	1354.56	1387069.25	10.00s
Digital Ocean	zap.	1KiB	970.48	993773.46	10.00s
AWS	naklj	1MiB	618.82	618.82	10.00s
Azure	naklj.	1MiB	411.51	411.51	10.00s
Digital Ocean	naklj.	1MiB	334.71	334.71	10.00s
AWS	zap	1MiB	15517.07	15517.07	6.60s
Azure	zap.	1MiB	8801.70	8801.70	10.00s
Digital Ocean	zap.	1MiB	7157.45	7157.45	10.00s



Slika 9.5: Grafi pisanja v pomnilnik.

Zanimala nas je tudi učinkovitost predpomnilnika pri posameznih ponudnikih. Večji kot je predpomnilnik, večja bi morala biti razlika med zaporednimi in naključnimi dostopi do pomnilnika. Iz slike 9.6 lahko vidimo, da po pričakovanjih pri zaporednem dostopu, količina prenešenih podatkov strmo naraste. Čeprav je absolutna razlika v prenosu med zaporednim in naključnim dostopom do pomnilnika največja pri AWSju, pa smo po podrobni analizi rezultatov ugotovili, da je razlika prenesešenih podatkov relativno največja pri Digital Ocean-u.



Slika 9.6: Graf testa lokalnosti (cache).

Po izvedenih bremenitvenih testih za delovni pomnilnik in analizi rezultatov smo prišli do zaključka, da je najboljši ponudnik na tem področju AWS. Tako Azure in Digital Ocean sta precej počasnejša, vendar sta si med sabo zelo konkurenčna.

9.6.3 Trdi disk

Meritve hitrosti diska smo izvedli za dve različni velikosti bloka, ker se pojavijo opazne razlike med hitrostmi prenosa večjih datotek in hitrostmi prenosa manjših datotek. Za simuliranje povprečne velikosti prenosa manjših datotek po navadi velja velikost 4KB, za večje pa 4MB. Zato smo vse meritve izvedli za ti dve velikosti. Vse meritve diska smo pognali 10. 5. 2017 v večernih urah. Meritve smo ponovili tudi kasneje in rezultati so se zanemarljivo razlikovali, zato nismo naredili nadaljnih ponovitev. Vse meritve smo izvedli neposredno na virtualki, v katero smo se povezali preko povezave SSH.

Meritve hitrosti diska in števila operacij na sekundo smo izvedli z ukazi, navedenimi v Listingu 9.5.

Listing 9.5: Ukazi za meritve hitrosti in števila operacij s programom *fio*.

```
#nakljucno branje majhnih datotek
fio --name=rand_read_4k --rw=randread --bs=4K --size=4G

#nakljucno branje velikih datotek
fio --name=rand_read_4m --rw=randread --bs=4M --size=4G

#nakljucno pisanje majhnih datotek
fio --name=rand_write_4k --rw=randwrite --bs=4K --size=4G

#nakljucno pisanje velikih datotek
fio --name=rand_write_4m --rw=randwrite --bs=4M --size=4G

#sekvencno branje majhnih datotek
fio --name=seq_read_4k --rw=read --bs=4K --size=4G

#sekvencno branje velikih datotek
fio --name=seq_read_4m --rw=read --bs=4M --size=4G

#sekvencno pisanje majhnih datotek
fio --name=seq_write_4k --rw=write --bs=4K --size=4G

#sekvencno pisanje velikih datotek
fio --name=seq_write_4m --rw=write --bs=4M --size=4G
```

Hitrosti branja in pisanja

Hipoteza: Vsi ponudniki zagotavljajo SSD hitrosti diskov. Pričakujemo, da bodo hitrosti branja in pisanja majhnih blokov manjše od 100 MB/s. Za velike bloke pa pričakujemo hitrosti nad 200 MB/s. Ponudniki naj bi nudili podobne hitrosti.

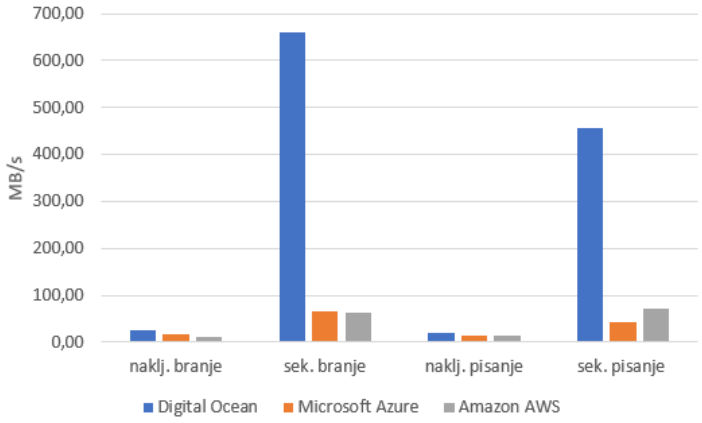
Tabela 9.7: Hitrosti branja in pisanja majhnih datotek (4KB) v MB/s.

Ponudnik	naklj. branje	sekv. branje	naklj. pisanje	sekv. pisanje
Microsoft Azure	17.45	65.32	14.51	44.06
Digital Ocean	25.07	659.48	19.11	455.61
Amazon AWS	12.27	61.44	14.83	72.02

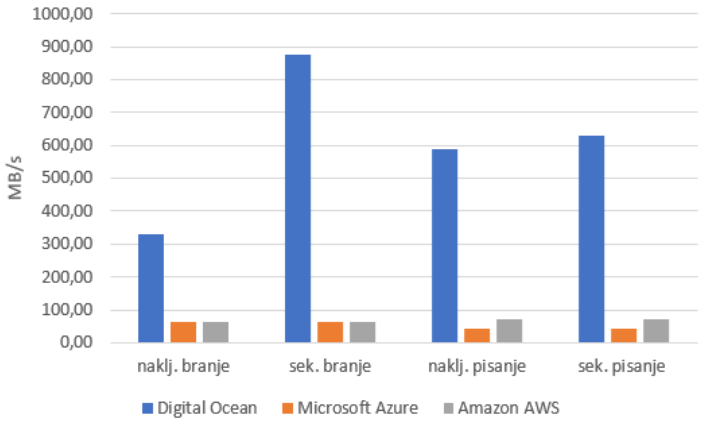
Tabela 9.8: Hitrosti branja in pisanja velikih datotek (4MB) v MB/s.

Ponudnik	naklj. branje	sekv. branje	naklj. pisanje	sekv. pisanje
Microsoft Azure	65.12	65.29	44.44	44.81
Digital Ocean	329.87	876.37	589.42	628.27
Amazon AWS	62.08	61.62	72.15	71.43

Komentar: Tabeli 9.7 in 9.8 prikazujeta izmerjene hitrosti branja in pisanja za vse ponudnike. Takoj lahko opazimo zelo veliko odstopanje ponudnika Digital Ocean, ki ponuja dejanske zmogljivosti SSD diska in močno prekaša oba druga ponudnika v skoraj vseh situacijah. Dohajata ga le pri naključnem branju in pisanju manjših datotek. Če medsebojno primerjamo še Azure in AWS, vidimo, da pri branju dajeta podobne rezultate, pri sekvenčnem pisanju in pisanju večjih datotek pa se AWS veliko bolje izkaže kot Azure. Izmerjene hitrosti pri ponudiku Digital Ocean dokazujejo, da ta ponudnik resnično vključuje zmogljivosti SSD diskov, medtem ko preostala dva dosegata le zmogljivosti klasičnih magnetnih diskov. Še bolj nazorno je vidna primerjava na slikah 9.7 in 9.8. Diska ponudnikov Azure in AWS sta torej v splošnem veliko počasnejša, kot smo predvideli v hipotezi. Le ponudnik Digital Ocean dosega rezultate, ki smo jih predvideli. Zanimiva ugotovitev, ki kaže, da sta deklaraciji ponudnikov Azure in AWS o SSD zmogljivostih nekoliko zavajajoči.



Slika 9.7: Hitrosti branja in pisanja majhnih datotek (4MB) v MB/s.



Slika 9.8: Hitrosti branja in pisanja velikih datotek (4MB) v MB/s.

IOPS

Hipoteza: Glede na deklaracije ponudnikov pričakujemo, da bodo dosegali podobne rezultate v istem velikostnem razredu.

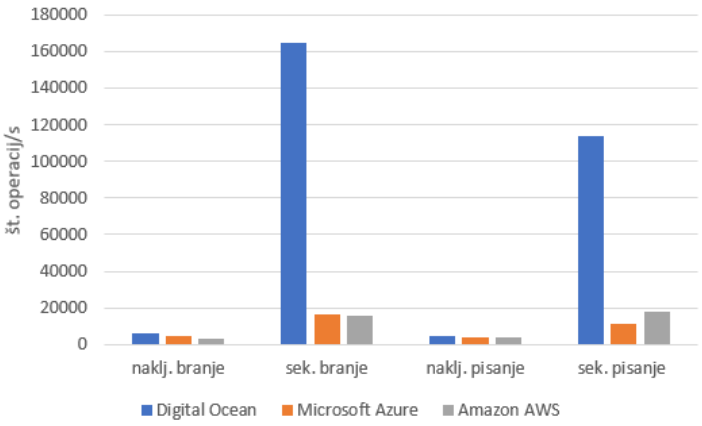
Tabela 9.9: Število operacij na sekundo pri branju in pisanju majhnih datotek (4KB).

Ponudnik	naklj. branje	sekv. branje	naklj. pisanje	sekv. pisanje
Microsoft Azure	4361	16329	3627	11016
Digital Ocean	6266	164870	4776	113901
Amazon AWS	3068	15361	3706	18005

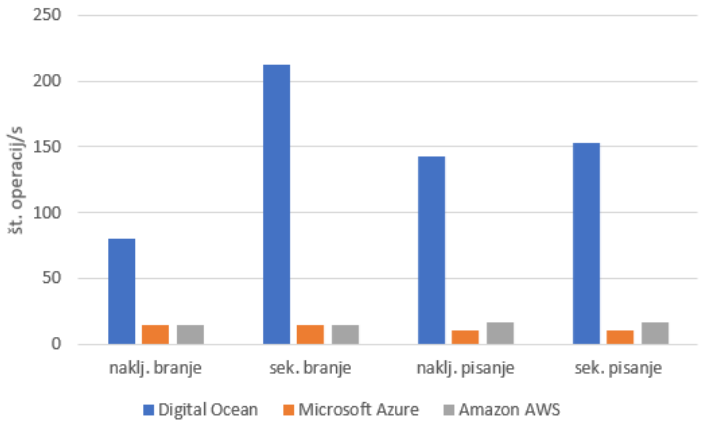
Tabela 9.10: Število operacij pri branju in pisanju velikih datotek (4MB).

Ponudnik	naklj. branje	sekv. branje	naklj. pisanje	sekv. pisanje
Microsoft Azure	15	15	10	10
Digital Ocean	80	213	143	153
Amazon AWS	15	15	17	17

Komentar: Tabeli 9.9 in 9.10 prikazujeta število možnih operacij na sekundo. Tudi v tej kategoriji je Digital Ocean v večini primerov za nekajkrat boljši od preostalih dveh. Rezutati vseh ponudnikov sledijo trendu zmogljivosti, ki smo ga videli že v prejšnji kategoriji. AWS tudi spet prekaša Azure po številu pisalnih operacij velikih datotek in sekvenčnem pisanju. Dobro grafično ponazoritev vidimo na slikah 9.9 in 9.10. Naše hipoteze niso bile točne zaradi precej velikih razlik med ponudniki.



Slika 9.9: Število operacij na sekundo pri branju in pisanju majhnih datotek (4KB).



Slika 9.10: Število operacij pri branju in pisanju velikih datotek (4MB).

Latenca

Latenco smo merili s programom *ioping* z ukazi na Listingu 9.6.

Listing 9.6: Ukazi za meritev latence s programom *ioping*.

```
# majhni bloki
ioping -s 4k -c 50

# veliki bloki
ioping -s 4m -c 50
```

Hipoteza: Glede na deklaracije ponudnikov ponovno pričakujemo, da bodo dosegali podobne rezultate v istem velikostnem razredu.

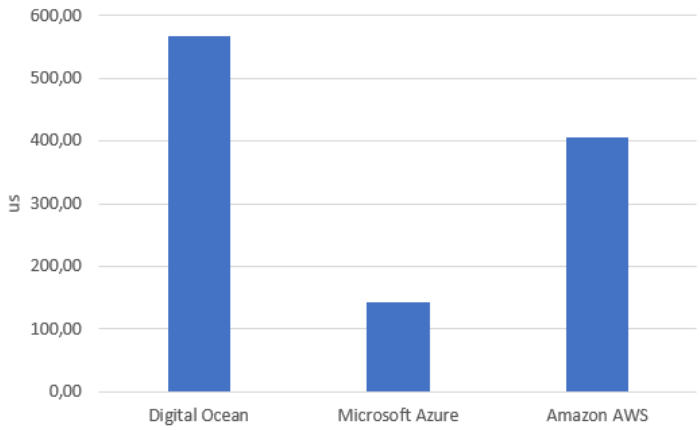
Tabela 9.11: Latenca pri dostopanju do majhnih datotek (4KB) v *us*.

Ponudnik	latenca [us]
Microsoft Azure	143
Digital Ocean	568
Amazon AWS	405

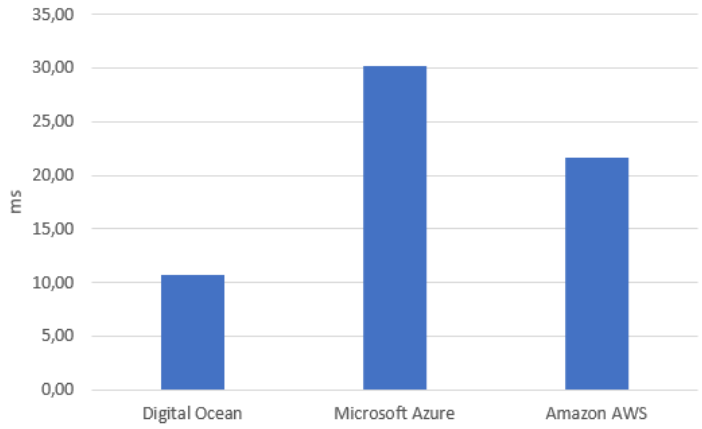
Tabela 9.12: Latenca pri dostopanju do velikih datotek (4MB) v *ms*.

Ponudnik	latenca [ms]
Microsoft Azure	30.20
Digital Ocean	10.70
Amazon AWS	21.6

Komentar: Tabeli 9.11 in 9.12 prikazujeta še latence diskov. Pri branju manjših datotek ima ponudnik Azure občutno boljšo latenco. Razlog za to je morda kakšna aktivnost v sistemu, ki je močno dvignila povprečje pri določenih meritvah drugih dveh ponudnikov. Pri večjih datotekah se spet bolje izkaže Digital Ocean, dvakrat bolje od AWS in trikrat bolje od Azure. Primerjava je prikazana na slikah 9.11 in 9.12. Odstopanje rezultatov od naših predvidenih hipotez spet dokazuje zelo različne zmogljivosti različnih ponudnikov.



Slika 9.11: Latenca pri dostopanju do majhnih datotek (4KB) v *us*.



Slika 9.12: Latenca pri dostopanju do velikih datotek (4MB) v *ms*.

Na koncu lahko iz rezultatov zaključimo, da edino Digital Ocean ponuja SSD zmogljivosti za dano naročnino, kar se izkaže v veliko boljših zmogljivostih diska. Presenetljiv zaključek, saj tudi druga dva ponudnika obljubljata zmogljivosti SSD diskov. Vendar se izkaže, da so zmogljivosti teh dveh ponudnikov le na nivoju magnetnih diskov. Je pa AWS veliko boljši od Azura pri pisanju večjih datotek in sekvenčnem pisanju.

9.7 Zaključek

Iz vseh pridobljenih podatkov lahko ugotovimo, da za zelo podobne cene pri vseh ponudnikih dobimo približno enake zmogljivosti procesorja. Zato sta odločujoča dejavnika zmogljivost diska in pomnilnika. Pri zmogljivosti pomnilnika sta Azure in Digital Ocean primerljiva, dosti boljši je AWS. Pri diskih pa je daleč najboljši Digital Ocean.

Če nam je najpomembnejši pomnilnik in nam zmogljivost diska ne predstavlja večjega pomena, se odločamo predvsem med Azurom in AWSjem. Digital Ocean pri pomnilniku ni najbolj konkurenčen predvsem zaradi občutno manjše količine. Tudi pri dvakrat dražji konfiguraciji ponuja še vedno manj kot trenutna konfiguracija Azura in le 250MB več kot trenutna konfiguracija AWS. Če se torej odločamo med AWS in Azure, je Azure verjetno boljša odločitev, kljub manjši zmogljivosti pomnilnika, saj ga ponuja dvakrat več. Poleg tega ponuja tudi skoraj štirikrat več diska.

V kolikor nam je pomembna zmogljivost diska, predvsem hitrost na nivoju SSDjev, se je najbolje odločiti za Digital Ocean, ki je veliko bolj zmogljiv od Azura in AWSja. Azure in AWS se tu izkažeta za nekoliko zavajujoča, saj zmogljivosti diskov niso dosegale modernih SSDjev. Če nam je pomembna le količina diska, je Azure spet najboljša izbira, saj ponuja dosti več od preostalih dveh ponudnikov.

Na podlagi naših meritev najmanj priporočamo AWS, saj je edino področje, na katerem je AWS dober, hitrost pomnilnika. Ta ni ravno najpomembnejši dejavnik, predvsem v primerjavi s količino pomnilnika. Smatramo, da je bolje imeti 50% počasnejši, a dvakrat večji pomnilnik. Poleg tega AWS ponuja izjemno malo diska. Če nam je najpomembnejša hitrost diska (SSD), je edina prava odločitev Digital Ocean. Za vse ostale primere priporočamo Azure, saj za le malo več denarja ponuja veliko več pomnilnika in diska ob primerljivi zmogljivosti procesorja.

Poglavje 10

Učinkovitost programske analize v oblaku

Žiga Sajovic

10.1 Opis problema

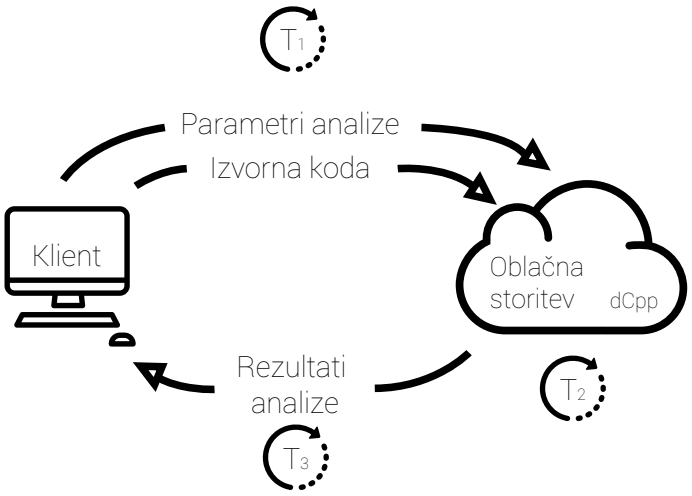
Ob naraščujoči potrebi po mobilnosti modernega človeka, se že dalj časa pojavlja trend selitev računalniških storitev v oblak. Potreba po mobilnosti je seveda v bitki s potrebo po učinkovitosti, misleč, da se vsota prednosti in slabosti sešteje v skupen prirastek učinkovitosti uporabnika.

Pod drobnogled vzamemo postavitev storitve analize programske kode v oblaku. V prid razumljivosti pričujočega besedila uvajamo nekaj definicij, ki so predstavljene na sliki 10.1:

- T_1 predstavlja čas prenosa podatkov s strani klienta na strežnik (tako izvorne kode, kot parametre analize);
- T_2 predstavlja čas izvajanja analize v oblaku, t.j. čas, ki je potreben za zahtevane izračune in analizo izvorne kode v oblaku;
- T_3 predstavlja čas prenosa rezultata s strani strežnika do uporabnika;

10.2 Namen

Strojna koda je že sama po sebi mnogokrat spominsko in procesno zahtevna, zato sledi enostaven sklep, da je analiza le-te še zahtevnejša. Delo programerja je mobilno, in v prid lahkotnosti je strojna oprema, ki jo programer nosi s seboj manj učinkovita. Selitev takšne storitve v oblak posledično predstavlja



Slika 10.1: Diagram storitve analize kode.

problem, ki terja da se razišče trg ponudnikov, ter analizira smiselnost umestitve takšne storitve v oblak. V ta namen testiramo zmogljivost oblaka, ob prisotnosti bremena analize kode in njegovo obnašanje ob višanju le-tega.

10.3 Izbira ponudnikov

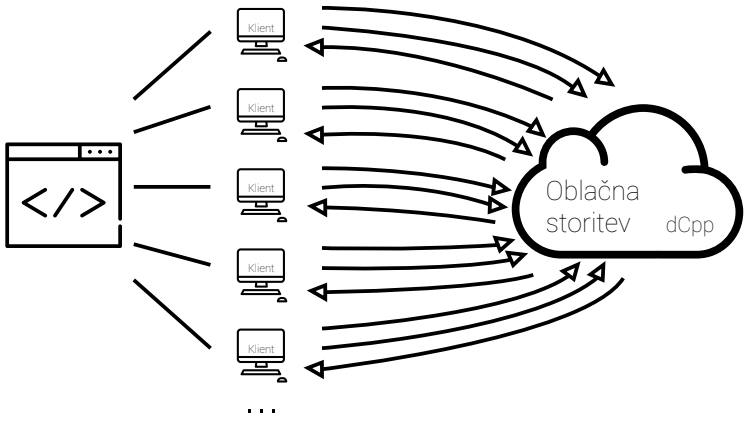
V pričujočem razdelku sta na kratko opisana ponudnika oblačnih storitev, katera smo želeli testirati in primerjati med seboj. Prvi od ponudnikov je razvojno okolje v oblaku *Cloud9*, ki omogoča testno vzpostavitev storitve, drugi pa *Heroku*, ki je "read-only"ponudnik storitve v oblaku (i.e. ustvarjanje datotek na serverju ni mogoče preko skript, temveč le ob vzpostavitvi storitve), ki omogoča vzpostavitev iz domačega računalnika.

10.4 Izbira tehnologij

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

10.4.1 Perf – Linux tools

Perf je orodje za performančno analizo v Linux okolju. Podpira pregled delovanja strojne opreme in izvajanje programske kode na njej. Orodja se bomo poslužili za merjenje razlik v učinkovitosti lokalno delujočih programov, kot tudi v oblaku.



Slika 10.2: Simulator bremena.

10.4.2 C++

C++ je programski jezik, ki omogoča nadzor nad nizko nivojskimi procesi in upravljanje s spominom.

10.4.3 Operacijski calculus na programskih prostorih

Operacijski calculus na programskih prostorih [100] je matematično podprta teorija, ki je plod avtorjevega raziskovalnega dela. Teorija omogoča zvezno obravnavanje programskih jezikov v matematično rigoroznem kontekstu.

dCpp

Knjižnica dCpp [101] je odprtokodna implementacija teorije Operacijskega calculusa na programskih prostorih [100], ki omogoča odvajanje programske kode (primer na sliki 10.3). Izvorno kodo spremlja članek Implementacija Operacijskega calculusa na programskih prostorih [102], ki vsebuje razlago delovanja in uporabe knjižnice. Delovanje knjižnice v oblaku z namenom analize programske kode bo predmet analize pričujočega poglavja.

10.5 Opis bremena

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

```
var foo(const var& x, const var& y)
{
    if(x < 1)
        return y;
    else if(y < 1)
        return x;
    else
        return x / foo(x / 2, y) + y * foo(x, y / 3);
}

dCpp::init(x);
dCpp::init(y);

var f = foo(x,y);
var df;
for(int i=0; i< 10; i++){
   df=f.d(&x);
}
```

Slika 10.3: Primer izračuna desetega odvoda funkcije foo po x.

dCpp

Knjižnica, ki ponuja storitev s katero bomo bremenili oblak, je natančneje predstavljena v predhodnem razdelku.

Simulator klientov

Spisan je simulator klientov (slika 10.2) odjemalcev, ki oblak vztrajno napadajo z zahtevami. Sam simulator omogoča konfiguracijo in manipulacijo tako frekvence napada, kot tudi težavnosti bremena.

Simulator vsebuje izvorne kode predvidene za analizo z orodjem dCpp, ki naraščajo po zahtevnosti in ceni v času in spominu. Izvajalo se bo več bremenitev hkrati z različnimi težavnostmi. To nam bo razkrilo korelacije med cenami storitev klientov. Denimo, da prvi klient na server odda izvorno kodo, ki je precej zahtevna, medtem ko drugi oddajo manjše bremenitve. Kako medsebojno vplivajo na čas čakanja odziva in dostavo rešitve?

10.5.1 Hipoteza zahtevnosti storitev

Naša hipoteza je, da je čas izvajajanja storitve v oblaku občutno daljši od bremena prenosa podatkov.

$$T_2 >> T_1 + T_3 \tag{10.1}$$

10.5.2 Hipoteza čakajočih

Predpostavljamo, da z naraščujočim številom uporabnikov in njihovih zahtev, narašča tudi T_2, saj povzroča zmanjšanje pretočnosti s čakajočimi zahtevami v vrsti. Drugače povedano, večje število uporabnikov pomeni daljši povprečen čas za prejetje odgovora.

10.5.3 Hipoteza odvisnosti

Postavljamo hipotezo, da cene storitev za kliente (v času in spominu) niso neodvisne med seboj (v kolikor so časovno lokalne).

Z U_i označimo zahtevo (v času in pomnilniku) uporabnika i. S T označimo funkcijo izračuna cene, ter s t označimo mejo, kjer velja, da je zahteva uporabnika U_i zahtevna če velja $T(U_i) > t$.

Določimo 2 množici uporabnikov, ki sta razbitje množice uporabnikov U po izrazu

$$U^1 = \{U_i : T(U_i) > t\}, \tag{10.2}$$

$$U^2 = \{U_i : T(U_i) \leq t\}, \tag{10.3}$$

kjer velja

$$U = U^1 \cup U^2 \text{ in } U^1 \cap U^2 = \{\emptyset\}. \tag{10.4}$$

Predpostavimo, da velja izraz

$$\sum_{U_i \in U} T(U_i) \neq T(\sum_{U_i \in U} U_i). \tag{10.5}$$

Hkrati predpostavimo da velja tudi izraz

$$c_{U^1} \cdot T(\sum_{U_i \in U^1} U_i) \implies c_{U^2} \cdot \sum_{U_i \in U^2} T(U_i). \tag{10.6}$$

V kolikor se poveča cena bremena prve skupine, to povzoroči narastek v ceni druge skupine, čeprav njihovo breme ostaja enako.

Predpostavljamo, da enakosti veljajo za čas T_2 in za čas T_3.

10.6 Zbiranje podatkov

V pričujočem razdelku opišemo način pridobivanja podatkov za analizo.

10.6.1 Čas do prejema zahteve T_1

V pričujočem razdelku obravnavamo čas trajanja od poslane zahteve s strani uporabnika do časa prejema zahteve na strežniku (i.e. T_1).

Postopek

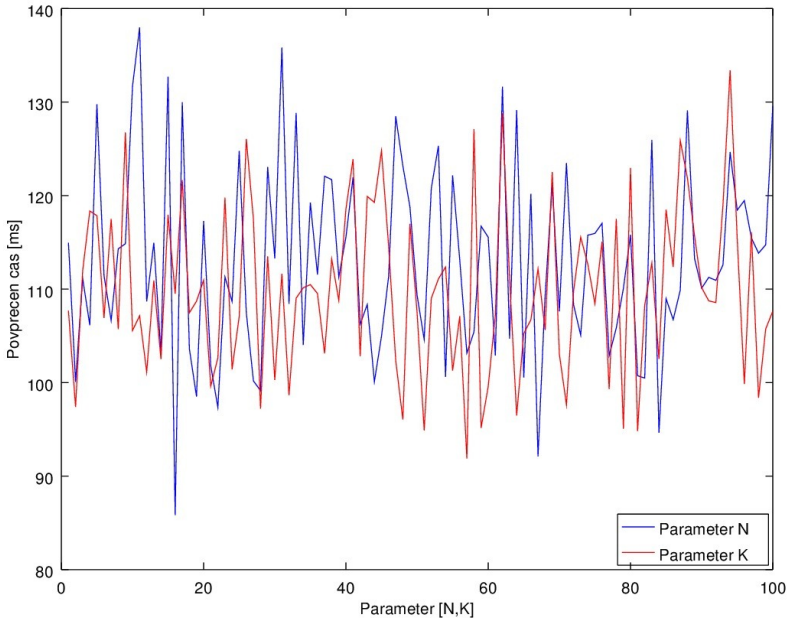
Za ustrezne meritve razlike v času, med točko, ko je sporočilo poslano in točko, ko je sprejeto na strani strežnika, smo uskladili lokalno uro s tisto na strežniku. S T_1^1 označimo čas poslanega sporočila in s T_1^2 čas prejetega sporočila na strežniku. Tedaj je čas T_1 enak

$$T_1 = T_1^2 - T_1^1. \tag{10.7}$$

Rezultati

V prid verodostojnosti testov, smo zaradi uporabe simulatorja pred pričetkom preverili, da so vsa pošiljanja znotraj zmožnosti naše upload linije.

Rezultati pri obeh ponudnikih se ujemajo, zato predstavljamo en poenoten rezultat. Sam čas pošiljanja je neodvisen od števila uporabnikov, ki hkrati dostopa do storitve, kar je razvidno iz slike 10.4, kjer z N označimo število uporabnikov, ki vsak naenkrat (v resnici zaporedno) odda po K programov v obdelavo. Prikazani so rezultati za pošiljanje datotek enakih velikosti. Čas pošiljanja ustrezno narašča ob večjih datotekah, kot gre pričakovati.



Slika 10.4: T_1 pri N in K.

10.6.2 Čas do prejema odgovora T_3

V pričujočem razdelku obravnavamo čas trajanja od zaključene analize na strani strežnika, ter do točke ko uporabnik prejme odgovor.

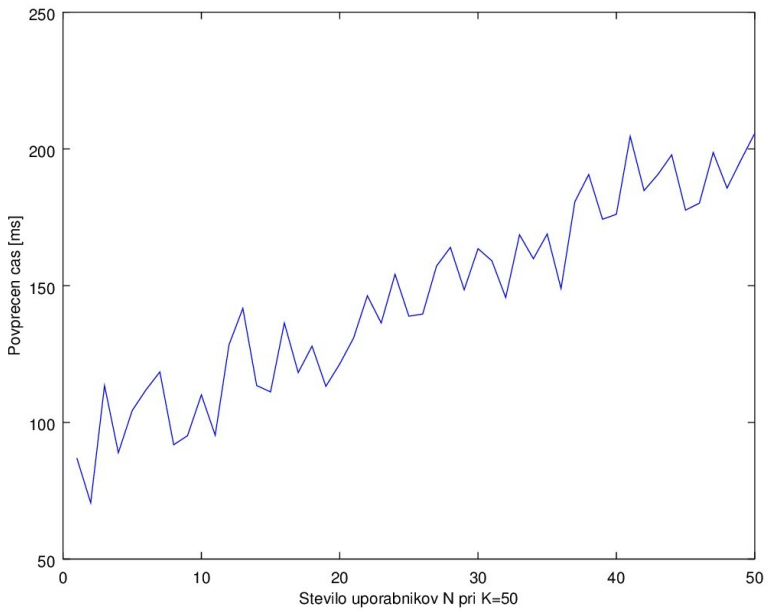
Postopek

Za ustrezne meritve razlike v času, med točko ko je sporočilo poslano in točko ko je sprejeto na strani strežnika, smo uskladili lokalno uro s tisto na strežniku. S T_3^1 označimo čas zaključka analize in s T_3^2 čas prejetega sporočila na strani uporabnika. Tedaj je čas T_3 enak

$$T_3 = T_3^2 - T_3^1. \tag{10.8}$$

Rezultati

V prid verodostojnosti testov smo zaradi uporabe simulatorja pred pričetkom preverili, da so vsa prejemanja znotraj zmožnosti naše download linije.



Slika 10.5: T_3 pri N in K.

Po predpostavkah razdelka 10.5.2 gre pričakovati povečevanje vrste čakajočih, ob povečevanju števila uporabnikov, ki storitev oblegajo z zahtevami.

```
for user in Users:
   for req in Requests:
      request.post(data(user));
   endfor
endfor
```

Slika 10.6: Pseudo koda testiranja osnovne odzivnosti.

Rezultati, ki so vidni na sliki 10.5 potrjujejo hipotezo. Opazimo, da čas v milisekundah narašča linearno v odvisnosti od števila uporabnikov, kar pomeni, da večje število uporabnikov doprinese k daljšemu povprečnemu času dostave sporočil.

10.6.3 Čas analize v oblaku T_2

S prvimi testiranji se spoznamo s storitvijo in ocenimo njeno odzivnost in točke odpovedi. S simulatorjem emuliramo N uporabnikov storitve, ki vsak naenkrat (v resnici zaporedno) odda po K programov v obdelavo.

Podatki so bili pridobljeni za oba ponudnika. Vsa zbiranja podatkov so se pričela z enakostmi bremena $N = 10$ in $K = 10$, višanja teh vrednosti pa so bila prilagojena posamičnem ponudniku.

Cloud9

V tem razdelku predstavljamo rezultate osnovne odzivnosti Cloud9 po postopku prikazanem na sliki 10.6, ki so prikazani v tabeli 10.1. Vrednosti v tabeli označujejo uporabnikov povprečen čas čakanja na odgovor storitve za posamičen program v sekundah.

	$K = 10$	$K = 20$	$K = 30$	$K = 50$
$N = 10$	5.88	5.95	4.96	7.06
$N = 20$	9.71	10.39	12.54	14.55
$N = 30$	12.01	13.07	11.62	17.43
$N = 50$	12.28	12.76	16.38	17.59

Tabela 10.1: Matrika osnovne odzivnosti storitve Cloud9.

Heroku

V tem razdelku predstavljamo rezultate osnovne odzivnosti Heroku po postopku prikazanem na sliki 10.6, ki so prikazani v tabeli 10.2. Vrednosti v tabeli označujejo uporabnikov povprečen čas čakanja na odgovor storitve za posamičen program, v sekundah.

	$K = 10$	$K = 30$	$K = 50$
$N = 10$	2.94	3.77	3.94
$N = 30$	10.3	10.4	10.7
$N = 50$	16.9	18.8	19.1
$N = 70$	18.2	21.5	20.0
$N = 100$	26.65	28.98	28.59

Tabela 10.2: Matrika osnovne odzivnosti storitve Heroku.

10.6.4 Zasedenost sistema

V pričujočem razdelku so predstavljeni pridobljeni podatki procesorske in pomnilniške zasedenosti. Pridobivanje podatkov je potekalo preko razpona nekaj ur in trajalo dokler se zasedenost ni umirila s standarnim odklonom manjšim od 1%.

Cloud9

Cloud9 pri dolgotrajnejših obremenitvah (npr. 10 min), ki so večje od $K = 10$ in $N = 10$ odpove in je takšne teste posledično nemogoče resno izvajati, saj storitev pri višjih vrednostih parametrov N in K odpove in server se preneha odzivati, ter uporabniku v odgovor vrača napako *Error 502: Bad Gateway*. Zato v nadaljevanju izpuščamo predstavitve pridobljenih rezultatov pri tem ponudniku.

Heroku

Zastonjskemu uporabniku pripada 7 CPU jeder, vsak z 2494Mhz. Predstavljamo rezultate meritev procesorske zasedenosti v odvisnoti od časa, pogojenih s številom uporabnikov N in številom programov K. Slika 10.7 predstavlja odstotek zasedenosti procesorja na ordinati in pretečene minute na abscisi.

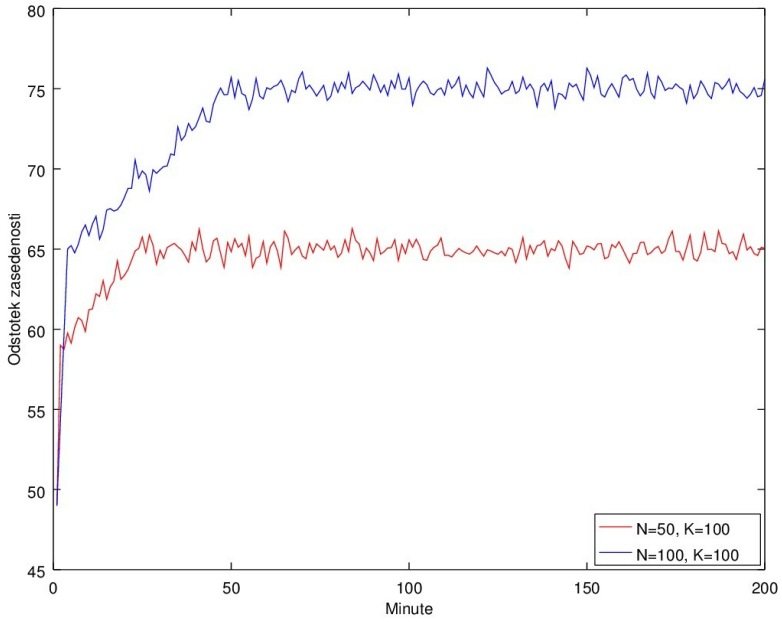
Analiza pričujočih podatkov na sliki 10.7 kaže, da procesorska zasedenost narašča linearno s časom do točke stabilnosti, kjer postane konstantna, kot je razvidno iz grafa. Pri parametrih $N = 50$, $K = 100$ se to zgodi pri procesorski zasedenosti 62% po pretečenem času $t = 25min$, pri parameterih $N = 100$, $K = 100$ pa pri 75% po pretečenem času $t = 50min$. Poudarjamo, da pri višjih vrednosti parametrov N in K (denimo > 150) storitev odpove.

Prav tako ugotavljamo, da je šum S porazdeljen normalno,

$$S : \mathcal{N}(0, \frac{1}{2}) \tag{10.9}$$

s povprečjem $\mu = 0$ ter odklonom $\sigma = \frac{1}{2}$.

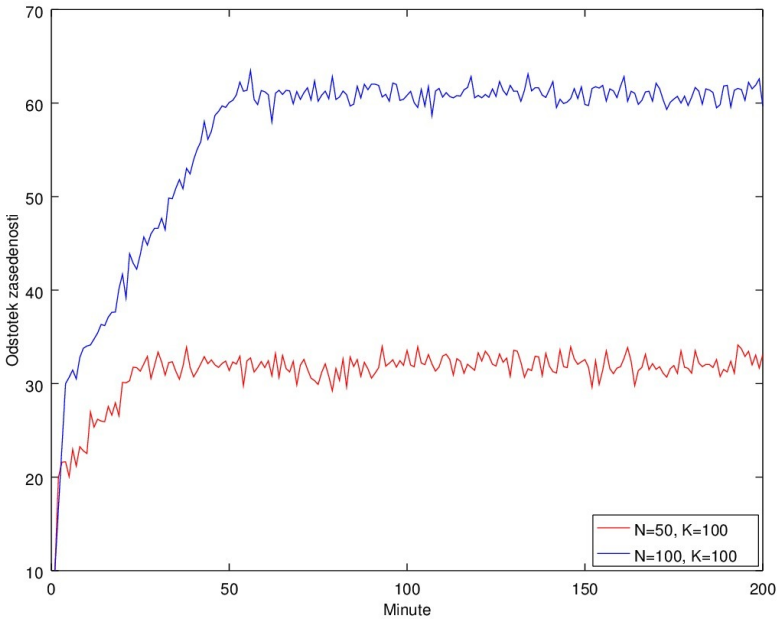
Dejstvo, da procesorska zasedenost precej časa narašča kaže na napako v sistemu. Pričakovano obnašanje bi bilo, da hitro preidemo v točko konstantne procesorske zasedenosti, saj je na serverju konstantno $N * K$ zahtev v obdelavi.



Slika 10.7: Odstotek zasedenosti procesorja za Heroku pri konstantnem N in K.

Preden raziščemo anomalijo pregledamo še pomnilniško zasedenost sistema, da pridobimo celotno sliko trenutnega stanja.

Brezplačnemu uporabniku pripada 512MB pomnilnika. Predstavljamo rezultate meritev pomnilniške zasedenosti v odvisnoti od časa, pogojenih na število uporabnikov N in število zahtev K. Slika 10.8 predstavlja odstotek zasedenosti pomnilnika na ordinati in pretečene minute na abscisi.



Slika 10.8: Odstotek zasedenosti pomnilnika za Heroku pri konstantnem N in K.

Analiza pričujočih podatkov na sliki 10.7 kaže, da procesorska zasedenost narašča linearno s časom do točke stabilnosti, kjer postane konstantna, kot je razvidno iz grafa. Pri parametrih $N = 50$, $K = 100$ se to zgodi pri pomnilniški zasedenosti 31% po pretečenem času $t = 25min$, pri parameterih $N = 100$, $K = 100$ pa pri 63% po pretečenem času $t = 50min$.

Prav tako ugotavljamo, da je šum S porazdeljen normalno,

$$S : \mathcal{N}(0, 0.9) \tag{10.10}$$

s povprečjem $\mu = 0$ ter odklonom $\sigma = 0.9$.

Počasno naraščanje pomniške zahtevnosti potrjuje sum napake v sistemu, saj bi spet šlo pričakovati hiter prehod v točko konstantne pomnilniške zasedenosti, saj je na serverju konstantno $N * K$ zahtev v obdelavi.

10.6.5 NodeJS in pomnilniške razpoke

Platforma NodeJS je spisana v programskem jeziku javaScript. To pomeni, da programer nima dostopa in nadzora nad upravljanjem s pomnilniškimi sredstvi, temveč je ta naloga prepuščena zbiralniku odpada (ang. *garbage collector*).

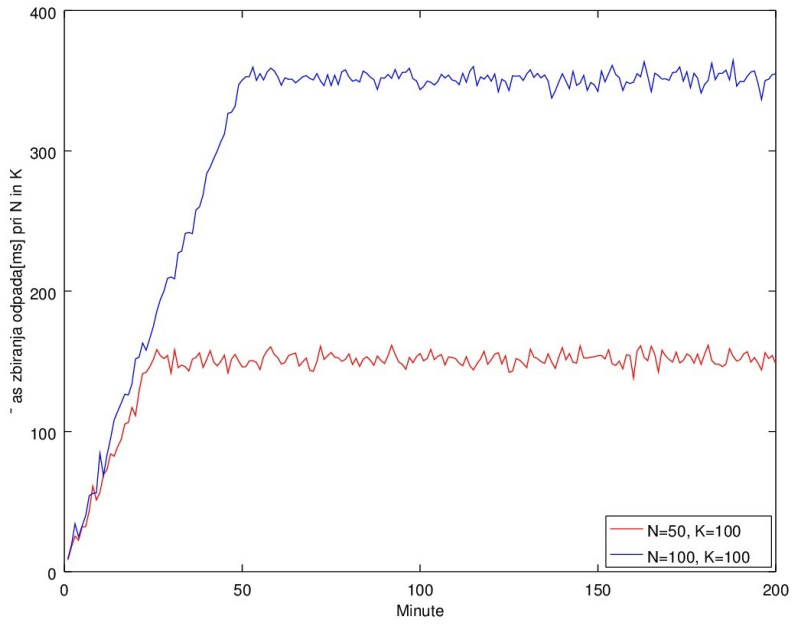
```
var restify = require('restify');

var server = restify.createServer();

var tasks = [];

server.pre(function(req, res, next) {
  tasks.push(function() {
    return req.headers;
  });

  req.user = {
    id: 1,
    username: 'Leaky Master',
  };

  return next();
});

server.get('/', function(req, res, next) {
  res.send('Hi ' + req.user.username);
  return next();
});

server.listen(3000, function() {
  console.log('%s listening at %s', server.name, server.url);
});
```

Slika 10.9: Ilustrativen primer aplikacije s pomnilniško razpoko.

Koncept česa razložimo na namernih anomalijah primera predstavljenega na sliki 10.6.5. Podatkovna struktura *tasks* se povečuje skozi celotno delovanje aplikacije in povzroči upočasnitev ter sčasoma odpoved aplikacije.

V dotičnem primeru obravnavane storitve predstavljenem na slikah 10.7 in 10.8 je zbiralec odpakov privedel do podobne anomalije. Zbiralec odpada uporablja strategijo *ustavi-svet*, kar pomeni, da več kot imamo objektov v pomnilniku, več časa bo zbiranje trajalo. Zato smo izvedli analizo časa, ki ga zbiralec odpada porabi med pobiranjem skozi čas. Rezultati so predstavljeni na sliki 10.10.

Analiza pričujočih podatkov na sliki 10.10 kaže, da čas trajanja zbiranja odpada narašča linearno s časom do točke stabilnosti, kjer postane konstantna, kot je razvidno iz grafa. Pri parametrih $N = 50$, $K = 100$ se to zgodi pri



Slika 10.10: Čas zbiranja odpada v *ms* za Heroku pri N in K.

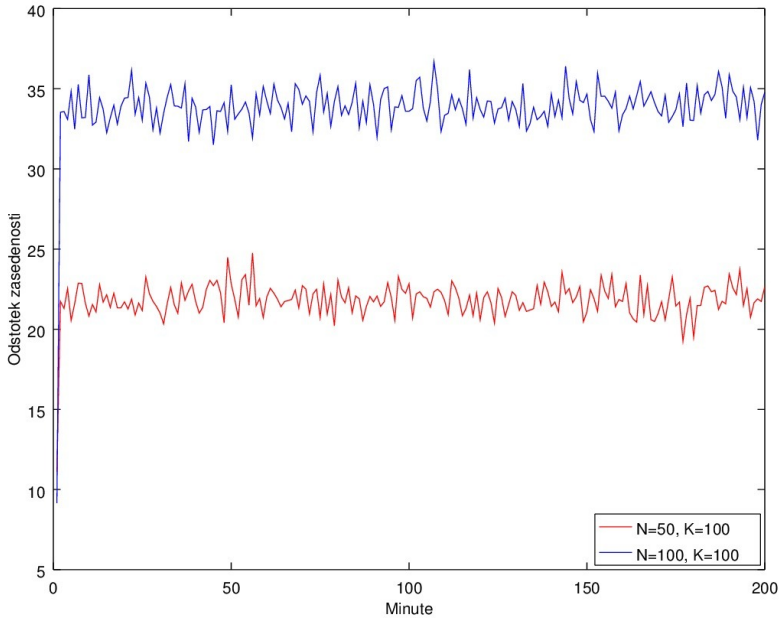
trajanju 151 ms po pretečenem času $t = 25$ *min*, pri parameterih $N = 100$, $K = 100$ pa pri 352 *ms* po pretečenem času $t = 50$ *min*.

Prav tako ugotavljamo, da je šum S porazdeljen normalno,

$$S : \mathcal{N}(0, 5) \tag{10.11}$$

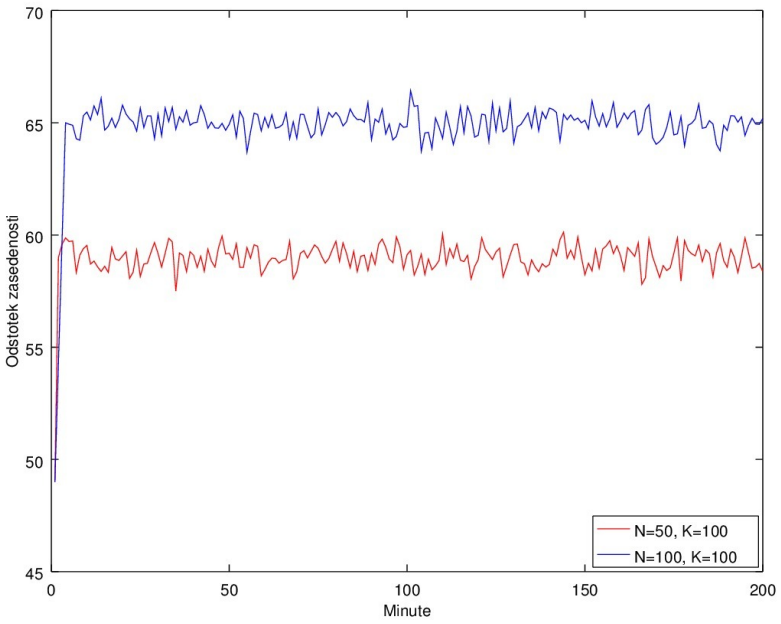
s povprečjem $\mu = 0$ ter odklonom $\sigma = 5$ms.

Ob vzporedni analizi procesorske zasedenosti (slika 10.7), pomnilniške zasedenosti (slika 10.8) in časa zbiranja odpada (slika 10.10) v vseh treh opazimo isti vzorec in isto točko stabilnosti. Posledično je naša predpostavka, da je zbiralec odpada krivec za počasno rast zasedenosti razpoložljivih sredstev. Zato smo vsa razporejanja s sredstvi, ki so bila spisana v *javaScript* vzpostavili na novo, kot zunanji program, ki ga NodeJS kliče po potrebi, sama pa nosi le nalogo pošiljanja pripravljene datoteke v odgovor. Na novo vzpostavljeni storitvi smo sprva ponovno izvedli teste pomnilniške zasedenosti.



Slika 10.11: Odstotek zasedenosti pomnilnika za Heroku pri N in K.

Rezultati (slika 10.11) se sedaj ujemajo s pričakovanji. Pomnilnik hitro doseže stabilno zasedenost, ki ostaja konstantna v času. Opazimo lahko da izgine ravno naraščanje zasedenosti, ki časovno sovpada z naraščanjem potrebnega časa za zbiranje odpada. To je razvidno ob primerjavi slike 10.8 s sliko 10.7, kjer opazimo zmanjšanje absolutne vrednosti razlike med krivuljama iz 30 *s* na 12 *s*, do česar privede zgoraj opisana razbremenitev sistema.



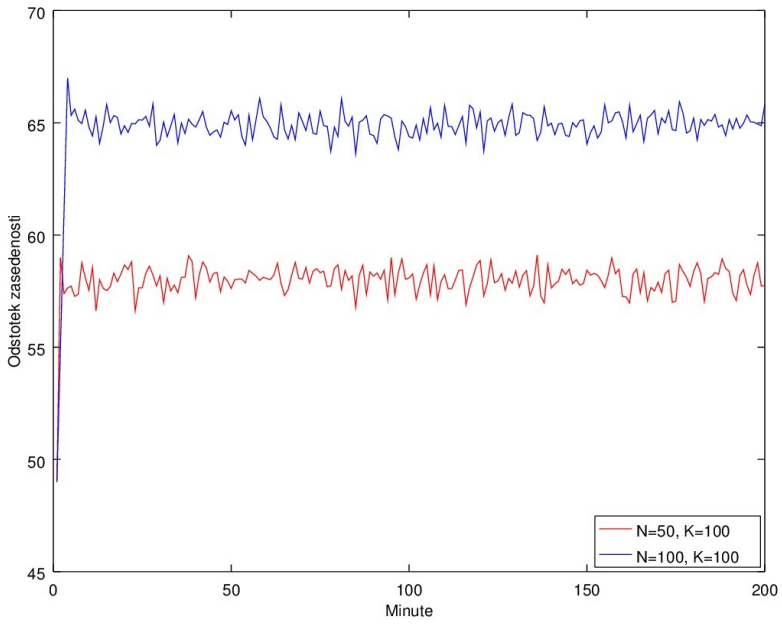
Slika 10.12: Odstotek zasedenosti procesorja za Heroku pri N in K.

Testiranje smo ponovili še za zasedenost procesorja in opazili identičen pojav. Rezultati (slika 10.12) se sedaj ujemajo s pričakovanji. Procesor hitro doseže stabilno zahtevnost, ki ostaja konstantna v času. Opazimo lahko da izgine ravno naraščanje zasedenosti, ki časovno sovpada z naraščanjem potrebnega časa za zbiranje odpada.

10.6.6 Rezultati

Ob razrešenih anomalijah lahko sedaj razmišljamo o zahtevah, ki jih terja storitev ob dolgotrajnem obratovanju. Storitev pri parametrih $N = 50$ in $K = 100$ zahteva 25% zasedenost pomnilnika in 59% zasedenost procesorja, medtem ko pri parametrih $N = 100$ in $K = 100$ zahteva 35% zasedenost pomnilnika in 68% zasedenost procesorja.

Hkrati ugotavljamo, da je pri zahtevnejših procesih izredno pomembno razpolaganje z razpoložljivimi sredstvi in da se ne gre slepo zanašati, da bo ali operacijski sistem ali zbiralnik odpada ustrezno in učinkovito skrbel za razpoložljiva sredstva sistema.



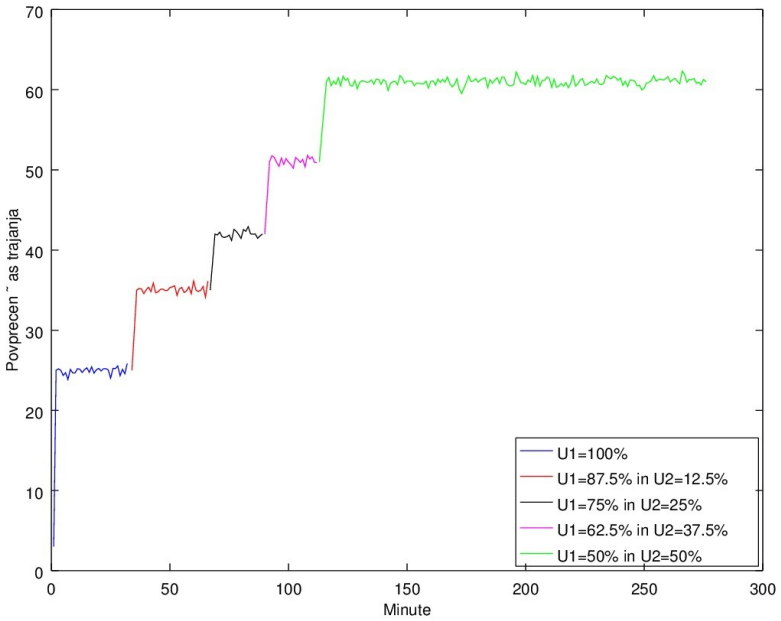
Slika 10.13: Graf razlike (zbiralec odpada) za Heroku pri N in K.

Na sliki 10.13 predstavljamo graf razlike med rezultati s slike 10.7, ter rezultati meritev procesorske zahtevnosti zbiralca odpada. Opazimo, da graf razlike sovpada z rezultati s slike 10.12 in potrjuje našo hipotezo.

10.7 Medsebojni vplivi uporabnikov

V pričujočem razdelku raziščemo medsebojne vplive uporabnikov, ki hkrati oddajajo zahteve z različno težavnostjo bremena. Vsi testi razdelka so izvedeni na ponudniku Heroku s parametri $N = 100$ in $K = 100$, torej, vsak od 100 uporabnikov oddaja 100 programov v obdelavo. Bralca opominjamo da U^1 pojmuje množico uporabnikov z bremenom standardne težavnosti in U^2 množico uporabnikov s težjim bremenom.

Testiranje izvajamo v štirih fazah. Začnemo s težavnostjo bremena s katerim smo pridobili podatke v tabelah 10.1 in 10.2. Tedaj so vsi uporabniki $u \in U^1$. Poženemo simulator in ga pustimo da izvaja testiranje, dokler čas izvajanje zahteve ne neha naraščati in preide v točko stabilnosti. Nato 12.5% uporabnikov $u \in U^1$ nadomestimo z uporabniki $u \in U^2$. Postopek iteriramo, dokler ni porazdelitev uporabnikov $u \in U^1$ in $u \in U^2$ ni $50\% - 50\%$.



Slika 10.14: Povprečen čas uporabnikov množice U^1 in $N = 100$, $K = 100$.

Rezultate predstavljamo na sliki 10.14, kjer prikazujemo povprečen čas trajanja storitve za uporabnike v množici U^1 (množice z lažjim bremenom).

10.7.1 Odvisnosti uporabnikov

Rezultati so predstavljeni na sliki 10.14. Ob postopku opisanem v prejšnjem razdelku smo merili povprečen čas odzivnosti storitve za uporabnike množice U^1. Spomnimo se hipoteze razdelka 10.5.3 in vsebujočih enakosti (10.5) in (10.6). Predpostavili smo, da bo povečevanje moči množice U^2 vplivalo na uporabniško izkušnjo uporabnikov množice U^1. Hipoteza je potrjena, kar je razvidno na sliki 10.14, saj kaže linearno naraščanje povprečnega časa U^1 vsakokrat ko povečamo moč množice U^2.

10.8 Verjetnostna analiza

V pričujočem razdelku predstavimo analizo verjetnosti odpovedi storitve, v odvisnosti od parametrov N in K. Podatke smo pridobivali podobno kot v prejšnjem razdelku, ter beležili število zavrnjenih storitev zaradi raznih razlogov, kot denimo preobremenjenosti pomnilnika in procesorja.

10.8.1 Cloud9

Cloud9 pri dolgotrajnešjih obremenitvah (npr. 10 *min*), ki so večje od $K = 10$ in $N = 10$ odpove in je takšne teste posledično nemogoče resno izvajati, saj storitev pri višjih vrednosti parametrov N in K odpove in server se preneha odzivati, ter uporabnikov v odgovor vrača napako *Error 502: Bad Gateway*. Posledično je za vse praktične primere verjetnost odpovedi 100%.

10.8.2 Heroku

V pričujočem razdelku predstavljamo verjetnostno analizo odpovedi storitve na zahtevo pri ponudniku Heroku pri vrednostih parametrov $N = 100$ in $K = 100$.

Verjetnost števila odpovedi

Število odpovedi je postopek, ki ga bomo modelirali z zveznim modelom, saj se zahteve lahko poslane kadarkoli v času (ki je zvezen pojav). Posledično Binomska porazdelitev ni ustrezna, temveč se poslužimo Poisonove porazdelitve, ki je rezultat limite Binomske, kjer število izbir pošljemo proti neskončnosti.

Poisonova porazdelitev je definirana z enačbo

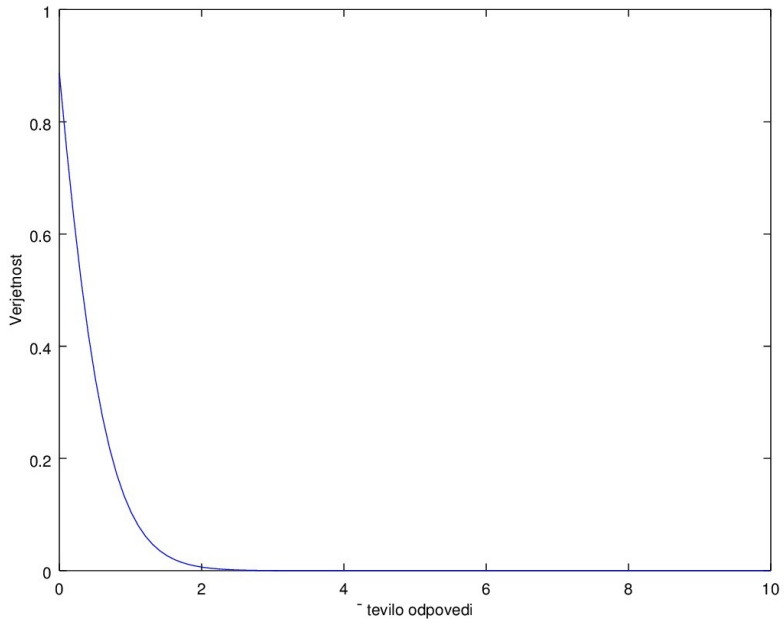
$$p(n) = \frac{\lambda^n}{n!} e^{-\lambda} \tag{10.12}$$

kjer je λ pričakovano število odpovedi na enoto v čakalni vrsti. Rezultati štetja odpovedi na enoto je pokazala, da velja izraz

$$\lambda = 0.12 \tag{10.13}$$

iz česar sledi

$$p(n) = \frac{0.12^n}{n!} e^{-0.12}. \tag{10.14}$$



Slika 10.15: Verjetnost števila odpovedi za $N = 100$, $K = 100$.

Pri težjih bremenih, (težjimi od U^2 (10.2)) za kakšne definiramo procese, ki terjajo $> 50Mb$ spomina, tudi pri Heroku pričnemo opažat podoben pojav, ki se pojavi pri Cloud9 že pri lažjih bremenih. Takrat se namreč pričakovano število odpovedi storitve na enoto pri $N = 100$ in $K = 100$ poveča na $\lambda = 99$.

Iz slike 10.15 je razvidno, da je ob $N < 100$ in $K < 100$ ter spominsko porabo $< 50MB$ na program, verjetnost za odpovedi več kot dveh zahtev skoraj ničelna (i.e. $P(2) = 0.006$), a verjetnost ene odpovedi je vseeno $P(1) = 0.1$. Verjetnost da do odpovedi sploh ne bi prišlo je $P(0) = 0.88$, kar kljub vsemu ni zadovoljivo.

Povečanje zanesljivosti sistema

V kolikor želimo povečati zanesljivost storitve, jo lahko vzporedno vežemo s sistemov, kjer zahtevo obravnava eden izmed njih. Predpostavljamo, da so naši sistemi identični in neodvisni (i.e. če odpove eden, to nikakor ne vpliva na druge). V ta namen definiramo par dogodkov E_i (dogodek, da storitev odpove pri vsaj eni zahtevi v vrsti i-tega sistema).

Spomnimo se enakosti (10.14) in izračunamo

$$P(E_i) = \sum_{n=1}^{\infty} \frac{\lambda^n}{n!} e^{-\lambda} = 1 - p(0), \tag{10.15}$$

$$\Longrightarrow$$

$$P(E_i) = 1 - e^{-0.12} = 0.11. \tag{10.16}$$

Zanima nas verjetnost $P(E^s)$ skupnega dogodka E^s (dogodek, da storitev ne odpove niti enkrat pri vsaj enemu od s sistemov)

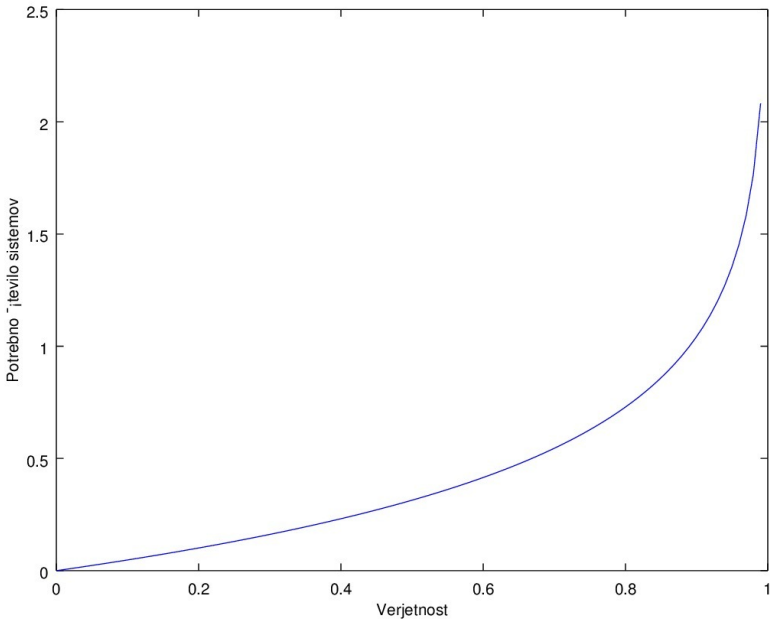
$$P(E^s) = 1 - \prod_{i=1}^{s} P(E_i), \tag{10.17}$$

$$\Longrightarrow$$

$$P(E^s) = 1 - 0.11^s. \tag{10.18}$$

Z enakostjo (10.18) je moč izračunat koliko vzporednih sistemov storitve s potrebujemo da dosežemo željeno verjetnost P'.

$$s = \frac{\ln(1 - P')}{\ln(0.11)} \tag{10.19}$$



Slika 10.16: Potrebno število sistemov za verjetnost ničelne odpovedi pri $N = 100$, $K = 100$.

Račun pokaže, da že pri treh vzporedno vezanih sistemih verjetnost naraste skoraj na 1 (99%), kot je razvidno iz slike 10.16.

Če se spomnimo Opombe **??**, kjer smo izpostavili drastično povišanje števila odpovedi, ko programi zahtevajo $> 50MB$ spomina, račun pokaže, da bi

v teh primerih potrebovali skoraj neskončno (i.e. MatLab vrača Inf) število vzporedno vezanih sistemov za zagotovitev iste verjetnosti (i.e. 99%).

Čas med odpovedmi

Skozi analizo pridobljenih podatkov je možno sklepati o pretečenem času med odpovedmi. Porazdelitev izpeljemo na podlagi naših predpostavk, da so dogodki odpovedi neodvisni in enako porazdeljeni. Ponovno z λ označimo pričakovano število odpovedi na izbrano časovno enoto ter definiramo $P(t < T)$ (verjetnost, da preteče T časa pred odpovedjo). Tedaj je verjetnost, da sistem ne odpove v časovnem obdobju Δt

$$P(T < t < T + \Delta t) = 1 - \lambda \Delta t, \tag{10.20}$$

kar zaradi predpostavke neodvisnosti pomeni

$$P(t < T + \Delta t) = P(t < T)(1 - \lambda \Delta t) \tag{10.21}$$

$$\Longrightarrow$$

$$\frac{P(t < T + \Delta t) - P(t < T)}{\Delta t} = -\lambda P(t < T). \tag{10.22}$$

Trenutki v času so lahko poljubno blizu (časovni intervali so lahko poljubno kratki), zato v enakosti (10.22) vzamem limito $\Delta t \to 0$, ki pokaže

$$\frac{dP(t < T)}{dt} = -\lambda P(t < T). \tag{10.23}$$

Izraz 10.22 predstavlja diferencialno enačbo katere edinstvena rešitev je

$$P(t < T) = e^{-\lambda t}. \tag{10.24}$$

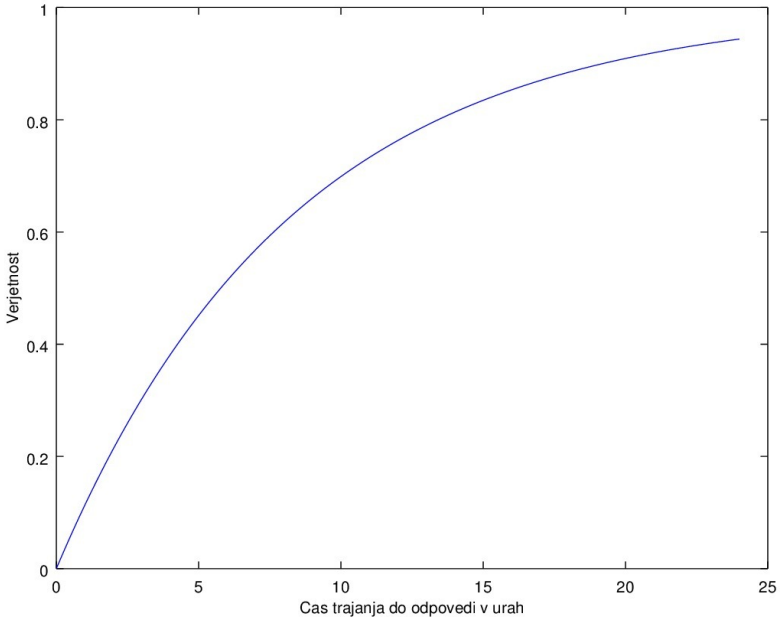
Dejstvo, da je gostota verjetnosti odvod mase (10.24) nas vodi v ugotovitev

$$p(t) = \lambda e^{-\lambda t}, \tag{10.25}$$

kar je iskana porazdelitev časa med odpovedmi, s katero poračunamo verjetnostno maso dogodka

$$P(t < T) = \int_0^T \lambda e^{-\lambda t} = 1 - e^{-\lambda T}. \tag{10.26}$$

Rezultati meritev so pokazali, da je za $N = 100$ in $K = 100$ in časovno enoto $1h$ pričakovano število odpovedi $\lambda = 0.12$, kar pomeni, da je pri stanju, kjer istočasno oddaja 100 uporabnikov po 100 programov v obdelavo, verjetnost, da vsaj enemu storitev odpove pri vsaj enem programu v določeni časovni enoti enaka 11% za eno uro in 94% za en dan. Rezultati so vidni na sliki 10.17.



Slika 10.17: Verjetnost da do odpovedi ne pride vsaj T časa pri $N = 100$, $K = 100$.

Povečanje zanesljivosti sistema

V kolikor želimo povečati zanesljivost storitve, lahko vzporedno vežemo s sistemov, kjer zahtevo obravnava eden izmed njih. Predpostavimo, da so naši sistemi identični in neodvisni (i.e. če odpove eden, to nikakor ne vpliva na druge). V ta namen definiramo par dogodkov, (E^s, T) (dogodek, da storitev ne odpove niti enkrat pri vsaj enemu od s sistemov v času T).

S podobnim sklepanjem kot pri enakostih (10.15) in (10.18) je moč dognati,

$$P(E^s) = 1 - \prod_{i=1}^{s} P(E_i) \tag{10.27}$$

$$\Longrightarrow$$

$$P(E^s, T) = 1 - (1 - e^{-0.12T})^s. \tag{10.28}$$

Z enakostjo (10.28) je moč poračunati koliko vzporednih sitemov storitve s potrebujemo da dosežemo željeno verjetnost P' v odvisnosti od željenega časa T.

$$s(T) = \frac{\ln(1 - P')}{\ln(1 - e^{-0.12T})} \tag{10.29}$$

Rezultate za različne vrednosti T predstavljamo na sliki 10.18.

10.9 Sklepi

V pričujočem razdelku predstavljamo sklepe, ki so rezultat opravljene analize.

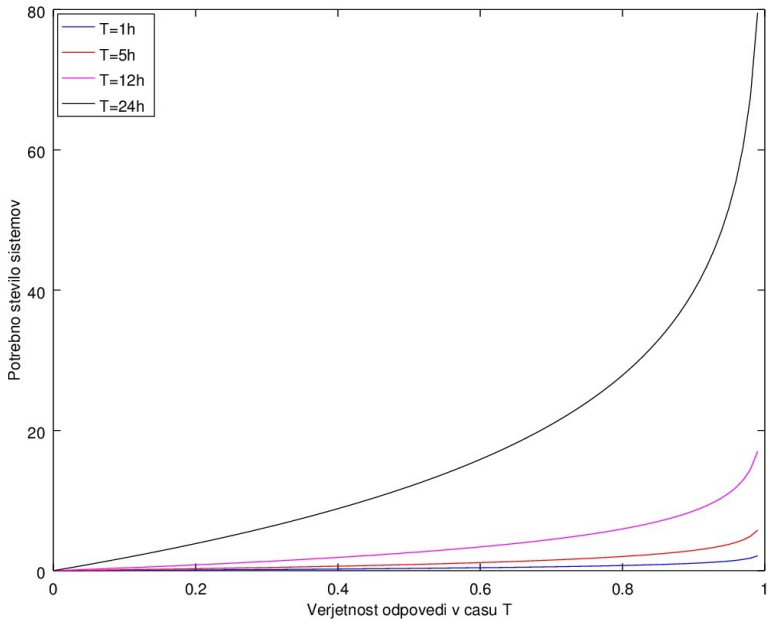
10.9.1 Ustreznost umestitve storitve v oblak

Analiza je pokazala, da je opisana storitev primerna za umesitev v oblak za vsakdanje potrebe večinskih uporabnikov (ki zahtevajo $< 50MB$ pomnilnika). A hkrati je pokazala, da je za analizo programske kode ogromnih projektov (kot so DeepLearning modeli in simulatorji obsegajočih fizikalnih procesov) pomnilniško in procesno prezahtevna za umestitiev v oblaku pri zastonjskih ponudnikih.

10.9.2 Primernost ponudnikov

Cloud9

Rezultati analize kažejo, da je ponudnik Cloud9 neprimeren za nosilca storitve, kar je smiselno, saj je prvotni namen ponudnika razvoj storitve v oblaku in ne nosilnost storitve same. Analiza je pokazala, da storitev, ki je nameščena v Cloud9 odpove že pri manjši zasedenosti in da je ponudnik kvečjemu primeren za prototipiranje storitve.



Slika 10.18: Potrebno število sistemov za verjetnost ničelne odpovedi v času T pri $N = 100$, $K = 100$.

Heroku

Rezultati analize kažejo, da je ponudnik Heroku primeren za nosilca storitve, četudi bo storitev gostila veliko hkratnih uporabnikov in zahtev. Pokazali smo, da je z vzporedno vezavo treh sistemov mogoče verjetnost odpovedi znižati skoraj na nič. A rezultati vseeno kažejo, da je ponudnik primeren le kot nosilec storitve z bremenom standardne težavnosti. Zagotovo veliko bremen, ki bi jih uporabniki terjali o storitve spada v to kategorijo težavnosti. A hkrati to pomeni, da storitev ne bi bila primerna za analizo projektov, katerih analiza terja $> 50MB$ pomnilnika, saj bi storitev uporabnikom s takšnim bremenom z veliko verjetnostjo odpovedala. Zato je storitev umeščena v oblak pri ponudniku Heroku neprimerna za uporabo s strani večjih podjetij, a dovolj zmogljiva za vsakdanje potrebe večine uporabnikov. Rezultati so bili pridobljeni na brezplačni različici storitve. Odpovedi, do katerih privedejo zahteve večje težavnosti ($> 50MB$) so najverjetneje posledica omejenih sredstev, ki so dodeljena brezplačnemu uporabniku. Ob vstopu v plačniško razmerje bi ponudnik uporabniku dodelil več sredstev in tedaj bi lahko zadovoljive rezutate pridobljene s testiranji z nižjo težavnostjo posplošili tudi na zahteve z večjo težavnostjo ($> 50MB$).

Literatura

[1] "Digitalocean: Simple cloud computing for developers." https://www.digitalocean.com/, Marec 2017.

[2] "Ted." http://www.theenergydetective.com/tedworks, Junij 2017.

[3] "energycurb." http://energycurb.com/, Junij 2017.

[4] "Mysql." https://www.mysql.com/, Marec 2017.

[5] "Mysqldb." http://mysql-python.sourceforge.net/MySQLdb.html, Marec 2017.

[6] "nmon." http://nmon.sourceforge.net/pmwiki.php, Maj 2017.

[7] "Raid." https://www.prepressure.com/library/technology/raid, Junij 2017.

[8] "Mqtt essentials." http://www.hivemq.com/mqtt-essentials/, April 2017.

[9] "Github education pack." https://education.github.com/pack, April 2017.

[10] "Digitalocean: Simple cloud computing for developers." https://www.digitalocean.com/, Marec 2017.

[11] "Amazon web services (aws) - cloud computing services." https://aws.amazon.com/, Marec 2017.

[12] "Pubnub chat solutions." https://www.pubnub.com/solutions/chat/, May 2017.

[13] "Mqtt malaria github repository." https://github.com/remakeelectric/mqtt-malaria, April 2016.

[14] S. Lee, H. Kim, D.-K. Hong, and H. Ju, "Correlation analysis of mqtt loss and delay according to qos level," *IEEE*, 2013.

[15] Z. B. Babovic, J. Protic, and V. Milutinovic, "Web performance evaluation for internet of things applications," *IEEEAccess*, 2016.

[16] "Vps (virtual private server), dedicated server, vps server, anti-ddos game, minecraft server, +120 localisatons, windows server linux - dedimax.com." https://www.dedimax.com/en/, Marec 2017.

[17] "Vps hosting | cheap windows linux virtual cloud server | 11." https://www.1and1.com/vps-hosting, Marec 2017.

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

[19] "Best dedicated servers vps in usa, nl russia." https://www.king-servers.com/en, Marec 2017.

[20] "Cheap windows linux vps hosting at an affordable price - contabo.com." https://contabo.com/?show=vps, Marec 2017.

[21] "Vps hosting | a managed virtual server solution for pros - godaddy uk." https://uk.godaddy.com/hosting/vps-hosting, Marec 2016.

[22] "Vps hosting | linux vps servers with cpanel - hostgator." http://www.hostgator.com/vps-hosting, Marec 2017.

[23] "Vps hosting | managed linux vps web hosting with cpanel - bluehost." https://www.bluehost.com/products/vps, Marec 2017.

[24] "Ssd vps | server ssd | vps.net." https://www.vps.net/products/ssd-vps, Marec 2017.

[25] "Node.js." https://nodejs.org/en/, April 2017.

[26] "Mongodb for giant ideas | mongodb." https://www.mongodb.com/, April 2017.

[27] "The leading operating system for pcs, tablets, phones, iot devices, servers and the cloud | ubuntu." https://www.ubuntu.com/, April 2017.

[28] "Details for europe /// internet traffic report." http://www.internettrafficreport.com/europe.htm, May 2017.

[29] "Data centers | cloud data center locations globally | vps.net." https://www.vps.net/cloud-datacenter-locations, May 2017.

[30] "The pros and cons of vps web hosting | web hosting hub." http://www.webhostinghub.com/web-hosting-guide/the-pros-and-cons-of-vps-web-hosting, May 2017.

[31] "Application programming interface." https://en.wikipedia.org/wiki/Application_programming_interface, Marec 2017.

[32] "Node.js." https://nodejs.org/en/, Marec 2017.

[33] "The go programming language." https://golang.org/, Marec 2017.

[34] "Digitalocean: Simple cloud computing for developers." https://www.digitalocean.com/, Marec 2017.

[35] "V8 javascript engine." https://en.wikipedia.org/wiki/V8_(JavaScript_engine), Marec 2017.

[36] "Node.js." https://www.npmjs.com/, Marec 2017.

[37] "Scala." https://en.wikipedia.org/wiki/Scala_(programming_language), Marec 2017.

[38] "Javavirutalmachine." https://en.wikipedia.org/wiki/Java_virtual_machine, Marec 2017.

[39] "Python." https://en.wikipedia.org/wiki/Python_(programming_language), Marec 2017.

[40] "Python pep." https://www.python.org/dev/peps/, Marec 2017.

[41] "Rest." http://www.restapitutorial.com/lessons/whatisrest.html, April 2017.

[42] "The infinite space between words." https://blog.codinghorror.com/the-infinite-space-between-words/, April 2017.

[43] "Dropbox: Dropbox cloud service." https://www.dropbox.com/tour/, March 2017.

[44] "Bash (unix shell)." https://en.wikipedia.org/wiki/Bash_(Unix_shell), April 2017.

[45] "Curl: curl - the man page." https://curl.haxx.se/docs/manpage.html, April 2017.

[46] "Dropbox: Dropbox developer blog - documenation for dropbox apis." https://blogs.dropbox.com/developers/, April 2017.

[47] "ping (networking utility)." https://en.wikipedia.org/wiki/Ping_(networking_utility), April 2017.

[48] "fallocate: fallocate(1) - linux man page." http://man7.org/linux/man-pages/man1/fallocate.1.html, May 2017.

[49] "Dropbox forum: Empty rtf file not downloadable through api." https://www.dropboxforum.com/t5/API-support/Empty-RTF-file-not-downloadable-through-API/td-p/54791, May 2017.

[50] "Dropbox forum: Dmca takedown notice on empty rar file." https://www.dropboxforum.com/t5/API-support/DMCA-Takedown-Notice-on-Empty-RAR-File/m-p/20080, May 2017.

[51] "Dropbox speed test." http://testmy.net/hoststats/dropbox, May 2017.

[52] "Software reviews: Synchronization test of dropbox, google drive, onedrive and livedrive." https://ergonotes.com/sync-test/, May 2017.

[53] "On the performance of cloud storage applications with global measurement." http://grid.hust.edu.cn/fmliu/iwqos16-CCS-fangmingliu.pdf, May 2017.

[54] "Inside dropbox: Understanding personal cloud storage services." http://www.tlc-networks.polito.it/oldsite/mellia/papers/DropboxImc12.pdf, May 2017.

[55] "Google Drive." Dosegljivo: `https://en.wikipedia.org/wiki/Google_Drive`. [Dostopano: 22. 3. 2017].

[56] "Google Drive." Dosegljivo: `https://www.google.com/drive/`. [Dostopano: 22. 3. 2017].

[57] "Google data center locations." Dosegljivo:`https://www.google.com/about/datacenters/inside/locations/index.html`. [Dostopano: 3. 5. 2017].

[58] "Mega (service)." Dosegljivo: `https://en.wikipedia.org/wiki/Mega_(service)`. [Dostopano: 22. 3. 2017].

[59] "Mega." Dosegljivo: `https://mega.nz/`. [Dostopano: 22. 3. 2017].

[60] "Dropbox: Celebrating half a billion users." Dosegljivo: `https://blogs.dropbox.com/dropbox/2016/03/500-million/`. [Dostopano: 1. 5. 2017].

[61] "Dropbox (service)." Dosegljivo: `https://en.wikipedia.org/wiki/Dropbox_(service)`. [Dostopano: 1. 5. 2017].

[62] "Dropbox data center locations." Dosegljivo:`http://www.eweek.com/storage/dropbox-expanding-into-europe-will-host-customer-data-in-germany`. [Dostopano: 3. 5. 2017].

[63] "AWS data center locations." Dosegljivo:`https://aws.amazon.com/about-aws/global-infrastructure`. [Dostopano: 3. 5. 2017].

[64] "C#." Dosegljivo: `https://sl.wikipedia.org/wiki/Programski_jezik_C_sharp`. [Dostopano: 12. 4. 2017].

[65] "Google Drive API." Dosegljivo: `https://developers.google.com/drive/`. [Dostopano: 30. 3. 2017].

[66] "Mega API." Dosegljivo: `https://github.com/gpailler/MegaApiClient`. [Dostopano: 12. 4. 2017].

[67] "Dropbox API Topics." Dosegljivo: `http://stackoverflow.com/documentation/dropbox-api/topics`. [Dostopano: 1. 5. 2017].

[68] "Dropbox API Explorer." Dosegljivo: https://dropbox.github.io/dropbox-api-v2-explorer/. [Dostopano: 1. 5. 2017].

[69] "Wireshark." Dosegljivo: https://en.wikipedia.org/wiki/Wireshark. [Dostopano: 3. 5. 2017].

[70] "The most popular cloud services rankings 2016." Dosegljivo: https://www.skyhighnetworks.com/cloud-security-blog/the-20-totally-most-popular-cloud-services-in-todays-enterprise/. [Dostopano: 10. 5. 2017].

[71] I. Drago, E. Bocchi, M. Mellia, H. Slatman, and A. Pras, "Benchmarking personal cloud storage," in *Proceedings of the 2013 conference on Internet measurement conference*, pp. 205–212, ACM, 2013.

[72] "Google API Client Library for .NET." Dosegljivo: https://developers.google.com/api-client-library/dotnet/reference/1.9.1/classGoogle_1_1Apis_1_1Upload_1_1ResumableUpload_3_01TRequest_01_4#ae8be093fb83c27dc10d27cbb8ce54c16. [Dostopano: 12. 4. 2017].

[73] "Dropbox Core API." Dosegljivo: https://www.dropbox.com/developers-v1/core/docs. [Dostopano: 4. 5. 2017].

[74] "Mega upload/download speed inconsistency." Dosegljivo: https://superuser.com/questions/95748/megaupload-download-speed-inconsistency. Dostopano: 11.05.2017.

[75] "File-hosting service comparison." Dosegljivo: http://www.principledtechnologies.com/Dropbox/Dropbox_for_Business_performance_comparison_1015.pdf. Dostopano: 25.05.2017.

[76] Wikipedia, "Heroku — Wikipedia, the free encyclopedia." http://en.wikipedia.org/w/index.php?title=Heroku&oldid=769173023, 2017. [Online; accessed 30-March-2017].

[77] Wikipedia, "Node.js — Wikipedia, the free encyclopedia." http://en.wikipedia.org/w/index.php?title=Node.js&oldid=772715861, 2017. [Online; accessed 30-March-2017].

[78] Postgres, "About postgressql." https://www.postgresql.org/about/, 2017. [Online; accessed 30-March-2017].

[79] M. Soldo, "Heroku postgres basic plan and row limit." https://blog.heroku.com/heroku_postgres_basic_plan_and_row_limits, 2012. [Online; accessed 30-March-2017].

[80] Heroku, "About postgres." https://devcenter.heroku.com/articles/heroku-postgresql, 2017. [Online; accessed 30-March-2017].

[81] Postgres, "Heroku postgres: Provisioning the add-on." https://devcenter.heroku.com/articles/heroku-postgresql#provisioning-the-add-on, 2017. [Online; accessed 6-April-2017].

[82] "Iskraemeco :: Domov." http://www.iskraemeco.com/si/. (Accessed on 05/24/2017).

[83] "Javascript - homepage." https://www.javascript.com/.

[84] "Node.js - documentation." https://nodejs.org/en/docs/.

[85] "C - programming language." https://en.wikipedia.org/wiki/C_(programming_language).

[86] "Python - documentation." https://www.python.org/doc/.

[87] "Enhanced network timing calculations for web response time metrics." https://www.ibm.com/support/knowledgecenter/SS5MD2_7.4.0.1/com.ibm.itcamt.doc/rt/Install_Guide/wrt_enhanced_network_timing.html.

[88] "Performance challenges in cloud computing." https://www.cmg.org/wp-content/uploads/2014/03/1-Paliwal-Performance-Challenges-in-Cloud-Computing.pdf.

[89] "Performance analysis of cloud computing centers." http://www.scs.ryerson.ca/~jmisic/papers/qshine2010hamzeh.pdf.

[90] "Performance evaluation methodology for cloud computing using data envelopment." https://www.thinkmind.org/download.php?articleid=icn_2015_3_20_30083.

[91] "Performance testing of cloud applications." http://www.remics.eu/system/files/REMICS_D6.6.lowres.pdf.

[92] "Amazon Web Services." Dosegljivo: https://aws.amazon.com. [Dostopano: 5. 4. 2017].

[93] "Microsoft Azure." Dosegljivo: https://azure.microsoft.com/en-us. [Dostopano: 5. 4. 2017].

[94] "Digital Ocean." Dosegljivo: https://www.digitalocean.com. [Dostopano: 5. 4. 2017].

[95] "sysbench." Dosegljivo: https://github.com/akopytov/sysbench. [Dostopano: 5. 4. 2017].

[96] "deflate." Dosegljivo: https://en.wikipedia.org/wiki/DEFLATE. [Dostopano: 9. 5. 2017].

[97] "phoronix-test-suite." Dosegljivo: https://wiki.mikejung.biz/Phoronix_Test_Suite. [Dostopano: 9. 5. 2017].

[98] "fio." Dosegljivo: `https://github.com/axboe/fio`. [Dostopano: 3. 5. 2017].

[99] "ioping." Dosegljivo: `https://github.com/koct9i/ioping`. [Dostopano: 3. 5. 2017].

[100] Žiga Sajovic and M. Vuk, "Operational calculus on programming spaces," 2016.

[101] Žiga Sajovic, "dcpp," 2016.

[102] Žiga Sajovic, "Implementing operational calculus on programming spaces for differentiable computing," *arXiv e-prints*, vol. arXiv:1612.0273, 2016.