# Analiza zmogljivosti oblačnih in strežniških storitev

Uredil prof. dr. Miha Mraz

Maj 2019

# 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 2018/2019 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 2019*

# Poglavje 1

# Testiranje zmogljivosti Amazon EC2 platforme

Peter Matičič, Jan Pelicon, Blaž Rojc

## 1.1 Opis problema

Med programerji je veliko takšnih, ki sanjajo o tem, da bi bili naslednji Bill Gates, Mark Zuckerberg ali Steve Jobs. Imajo idejo, za katero verjamejo, da bo zavzela svet in jim prinesla miljone ter večno slavo. Ampak potrebujejo platformo, na kateri bo njihova storitev tekla. En sam prenosnik ne more vendar streči tisočem uporabnikom s celega sveta hkrati. Platforma mora biti cenovno dostopna, hkrati pa tudi poljubno razširljiva, da v primeru, ko se zgodi neizogiben povečan obisk uporabnikov, lahko programer enostavno in hitro aktivira dodatno procesno moč.

Tu nastopi Amazonov Elastic Cloud [1]. Obljublja dostopne cene, fleksibilno alokacijo računskih virov in za nadobudnega podjetnika najpomembnejša možnost uporabe določenih storitev brezplačno. Med temi storitvami je na voljo tudi najem tako imenovanih "mikro instanc" [2]. To so virtualni spletni strežniki z enim procesnim jedrom in 1 GB pomnilnika [3]. Predstavljajo minimalno konfiguracijo, ki lahko gosti poljubno spletno storitev, hkrati pa predstavljajo tudi procesno ozko grlo, katerih omejitve moramo upoštevati pri postavitvi storitve.

S tem v mislih želimo stestirati platformo Amazon EC2. Ustvarili bomo enostavno storitev, ki bo uporabniku omogočala iskanje vzorcev v večjem naboru slik, shranjenih v oblaku. Predstavljala bo generično spletno aplikacijo, ki potrebuje ravno dovolj računske moči, da se bodo pojavile slabosti mikro instanc v obliki upočasnjenega ali onemogočenega delovanja na strani uporabnika. Osnova storitve je iskanje vzorca v naboru slik. Uporabnik od storitve zahteva podatek o tem, v katerih slikah se ta vzorec nahaja, pričakuje pa hiter

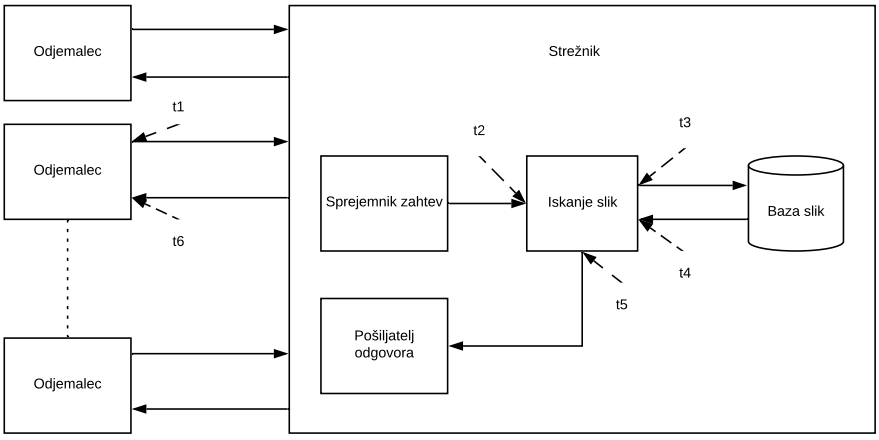
odgovor, z ne več kot nekaj sekundami zamika. Taka storitev nam bo omogočala relativno enostavno merjenje odzivnosti platforme, modularnost nalaganja kode in morebitno razširljivost v primeru večjega števila hkratnih uporabnikov.

## 1.2 Realizacija

Storitev je sestavljena iz dveh delov, in sicer strežnika na strani storitve EC2 in odjemalca na lokalnem računalniku. Opazujemo odzivnost strežnika, tako z meritvami na strežniku samem, kot pri odjemalcu.

### 1.2.1 Opazovano okolje

Aplikacija je realizirana kot spletna storitev, t.j. strežnik, dostopen na spletu, ki se odziva na zahteve uporabnikov. Zasnovana je po shemi na sliki 1.1. Uporabnik strežniku pošlje zahtevo v obliki JSON niza, ki vsebuje zaporedno številko zahteve, podatek o tipu zahteve in potrebne parametre. Strežnik zahtevo primerno obdela in vrne rezultat v obliki JSON niza, katerega oblika je odvisna od tipa zahteve.




Slika 1.1: Shema opazovane storitve.

### 1.2.2 Tipi zahtev

V osnovi vsi tipi zahtev vključujejo iskanje vzorca v naboru slik. Razlikujejo se v tipu vzorca, ki ga uporabnik želi najti v sliki. Storitev nudi tri tipe iskanj:

- iskanje specifične barve piksla,
- iskanje piksla, podobnega specifični barvi,
- iskanje slikovnega izseka.

### Iskanje specifične barve piksla

Storitev prejme podatek o iskani barvi piksla. Zaporedoma preiskuje slike v oblačni podatkovni bazi. Takoj ko najde prvo pojavitev iskanega piksla, iskanje ustavi in vrne podatke o tej pojavitvi - zaporedno številko slike in koordinati $x$ ter $y$ najdenega piksla. Če ne najde nobene pojavitve, preišče vse slike v podatkovni bazi in vrne sporočilo, da piksel ni bil najden.

### Iskanje piksla, podobnega specifični barvi

Poleg iskane barve storitev prejme od odjemalca še največje dovoljeno odstopanje. Zaporedoma preiskuje slike v podatkovni bazi, dokler ne najde prvega piksla, katerega barva se od iskane po komponentah razlikuje za največ toliko, kot določa odstopanje. Razlika se izračuna po enačbi

$$\begin{aligned} Razlika(RGB_{piksel}, RGB_{iskan}) = \\ = |R_{piksel} - R_{iskan}| + |G_{piksel} - G_{iskan}| + |B_{piksel} - B_{iskan}|. \end{aligned} \quad (1.1)$$

Če je iskana barva naključno izbrana, potem lahko za dano odstopanje izračunamo približno verjetnost, da bomo dobili ujemanje z naključnim pikslom. Za vsako barvo in odstopanje $n$ obstaja največ $\frac{1}{3}(n^3 + 6n^2 + 14n + 3)$ barv, ki se od iskane barve po komponentah razlikujejo za $n$. Vsak barvni kanal lahko zavzema vrednosti od 0 do 255, torej je za izbiro barve na voljo $2^{24} = 16777216$ vrednosti. Verjetnost aproksimiramo kot $\frac{n^3+6n^2+14n+3}{3 \cdot 16777216}$. Nekaj verjetnosti zadetka in ustreznih odstopanj je prikazanih v tabeli 1.1.

| verjetnost | 0.05% | 0.1% | 0.2% | 0.5% | 1% | 2% | 5% | 10% |
|---|---|---|---|---|---|---|---|---|
| odstopanje | 18 | 23 | 29 | 39 | 50 | 63 | 85 | 107 |

Tabela 1.1: Verjetnosti zadetka in njihova odstopanja

Ampak piksli v slikah niso naključni. Boljšo oceno dobimo, če privzamemo, da je barva cele slike *približno* enaka - če ne najdemo ustreznega piksla na začetku slike, potem je velika verjetnost, da ga tudi v preostanku slike ne bomo. Tako lahko izračunamo, koliko slik bo v povprečju pregledanih s pomočjo matematičnega upanja po formuli

$$E(p) = \sum_{i=1}^{\text{število slik}} i \cdot (1-p)^{i-1} \cdot p = \frac{1}{p}. \quad (1.2)$$

To pomeni, da bomo pri verjetnosti 1% v povprečju pregledali približno 100 slik pri vsaki zahtevi. Izkaže se, da je ta ocena v primeru te storitve kar dobra.

**Iskanje slikovnega izseka**

Storitev prejme slikovni izsek. Zaporedoma preiskuje slike v podatkovni bazi, dokler ne najde prvega ujemanja podanega izseka z izsekom na neki sliki. Izseka se ujemata, če je barva vsakega piksla danega izseka za največ 7 oddaljena od barve ustreznega piksla na sliki. Razlika barv se računa tako kot pri podobni barvi piksla (enačba 1.1). Odstopanje je tu potrebno le zato, ker pri shranjevanju izsekov pride do artefaktov stiskanja [4].

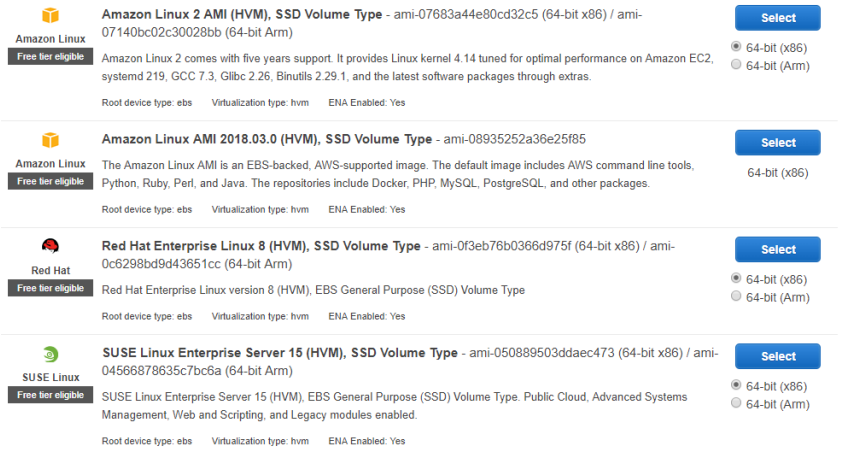
### 1.2.3 Slikovni material

Za potrebe zgoraj naštetih metod in njihovega testiranja smo na strežnik naložili 540 slik v JPEG formatu, ki smo jim zmanjšali velikost na 300 pikslov po širini in 400 pikslov po dolžini ali obratno za hitrejše procesiranje. Hkrati nam to zagotavlja, da je časovna zahtevnost primerjave približno enaka za ves slikovni material. Prav tako je bil cilj zbrati slike s čim večjo vsebinsko raznolikostjo, saj je to pomembno pri iskanju barve piksla in iskanju piksla podobnega specifični barvi. Nekaj primerov slik se nahaja na sliki 1.2.

Slika 1.2: Nekaj primerov slik iz nabora.

### 1.2.4 Amazon Web Services (AWS)

Za delo z Amazon Web Services [5] si moramo ustvariti račun v njihovem sistemu. Po uspešni registraciji si lahko ustvarimo svojo instanco EC2 storitve, ki nam jo Amazon ponuja zastonj za eno leto, pri čemer lahko na mesec porabimo največ 750 ur delovanja ponujenih instanc. Obsežnejša navodila so na voljo tudi na Amazonovi spletni strani [6], mi pa to naredimo tako, da se postavimo v AWS Management Console [7], kjer lahko izberemo opcijo *Launch a virtual machine with EC2*. Takoj nam konzola ponudi izbiro *slike navideznega diska* - vnaprej priravljenega nabora datotek, ki ga lahko neposredno zaženemo na instanci [8]. Izbor možnih slik diska je prikazan na sliki 1.3. Za naš projekt izberemo sliko Amazon Linux 2 AMI c.




Slika 1.3: Slike navideznih diskov za izdelavo virtualke.

V naslednjem koraku izberemo tip instance slike, to je Amazonov način izbire paketov, ki vključujejo različne funkcionalnosti. Ker v našem primeru izbiramo brezplačno različico, nam ponujajo tip t2.micro, ki vsebuje 1 jedro, 1GB pomnilnika, nizko prioriteto hitrosti povezave do same instance strežnika (umetno omejena hitrost s spremenljivo največjo hitrostjo) in hrambo podatkov na platformi Elastic Block Storage, ki nam ponuja brezplačno hranjenje podatkov na mrežno povezani shrambi [9]. Za tem lahko nadaljujemo z nastavljanjem različnih konfiguracij naše virtualke ali pa preprosto kliknemo Review and Launch, ki nam ponudi še en pregled izbranih nastavitev in zažene virtualko. Po zagonu virtualke nam sistem ponudi opcijo generiranja para ključev za varno SSH povezavo do nje. Ko zaključimo z ustvarjanjem, se premaknemo v EC2 management console kjer kliknemo na *instances*. Od tam lahko opazujemo status naše storitve in pridobimo tudi naslov, na katerem se nahaja. Za povezavo uporabimo javni naslov storitve, uporabnika $ec2 - user$, za varnost pa uporabimo *.pem* datoteko (Privacy Enhanced Mail Security Certificate), ki smo jo v prejšnjem koraku prenesli. Ko se uspešno povežemo na storitev, lahko pričnemo z razvojem naše aplikacije.

### 1.2.5 Aplikacija

Aplikacija je napisana v programskem jeziku Java, ki je interpretirana preko sprotnega prevajalnika, kar se lahko potencialno izkaže kot ozko grlo. Sestavljata jo odjemalska in strežniška komponenta, ki uporabljata skupne enumeratorje za določanje tipa zahteve. Zahteve sestavlja odjemalec in jih pošlja strežniku, ki te zahteve obdela - začne iskanje v slikah in pripravi prvi najden rezultat. Povezava med strežnikom in odjemalcem je trajna, dokler je eden od njiju ne prekine, kar nam omogoči, da pri meritvah ne upoštevamo časa vzpostavitve povezave. Podatki se prenašajo v obliki JSON, ki vsebuje sekvenčno številko zahteve, tip zahteve in morebitne dodatne podatke, ki so potrebni za obdelavo.

V izpisu 1.1 je predstavljen primer zahteve v obliki dokumenta JSON, ki jo odjemalec pošlje strežniku.

```
{
   "reqId": 1,
   "reqType": "PIXEL_NEAR",
   "pixelValue": "0xFFFA3881",
   "maxDistance": 86,
   "image": "",
   "req_start": 1555162220550,
   "err": ""
}
```

Listing 1.1: Primer JSON zahteve.

Strežnik ob prejemu podatkov začne z delom na ustrezni zahtevi ter nato pošlje odgovor klientu s številko zahteve in rezultatom. Odgovor vsebuje tudi čas proceiranja in branja slik. Strežniški del je napisan tako, da lahko paralelno obdeluje več zahtev. V izpisu 1.2 je predstavljen primer odgovora na zahtevo, ki ga strežnik vrne odjemalcu.

```
{
   "reqId": 1,
   "imageId": 4,
   "location": {
      "x": 124,
      "y": 87
   },
   "proc_time": 4528,
   "req_start": 1555162220550,
   "image_fetch_time": 78,
   "err": ""
}
```

Listing 1.2: Primer JSON odgovora.

## 1.3 Uporabljene metrike

Kot glavno metriko bomo opazovali skupni čas zahteve in odgovora.

Podrobneje ga bomo razdelili na čas dostopa do datotečnega sistema (zbirke slik v mapi na datotečnem sistemu, recimo ji baza slik) ($t_{baza} = t_4 - t_3$) in celoten čas obdelave na strežniku ($t_{streznik} = t_5 - t_2$). Hkrati merimo celoten čas trajanja zahteve ($t_{zahteva} = t_6 - t_1$) (časi označeni na sliki 1.1). Z izmerjenimi časi lahko izračunamo tudi druge kot sta čas procesiranja na strežniku ($t_{procesiranje} = t_{streznik} - t_{baza}$) in čas paketa na mreži ($t_{mreza} = t_{zahteva} - t_{streznik}$). S tem smo se znebili problema sinhronizacije ur med odjemalcem in strežnikom. Če bi želeli meriti tudi čas potovanja paketa od od-

jemalca do strežnika in čas potovanja paketa od strežnika do odjemalca, pa bi potrebovali tudi sinhronizacijo ur, ker pa naš cilj ni meriti čase potovanja paketov, saj je to namreč lastnost omrežja in ne same platforme, bomo zadovoljni s skupnim časom paketa na omrežju.

Zanima nas, kako se storitev odziva na zahteve ob različnih urah. Za potrebe meritev bomo odziv sistema merili ob treh različih časih v dnevu. Želimo izvedeti tudi, kako se časi odgovorov podaljšajo glede na število hkratnih uporabnikov.

## 1.4 Meritve

Zmogljivost storitve smo merili večkrat. Testirali smo hitrost iskanja podobne barve, opazovali smo razlike v hitrosti odziva iz različnih lokacij in ob različnih časih.

### 1.4.1 Poskusno testiranje

V soboto, 27. aprila 2019 ob 20.37 smo poskusno testirali iskanje piksla v okolici barve. Strežniku je bilo poslanih 500 zahtev, vsaka je vsebovala naključno izbrano barvo, strežnik pa je iskal piksel z odstopanjem, manjšim ali enakim 50. Zahteve so bile poslane zaporedno, kar pomeni, da je po oddaji zahteve odjemalec počakal na odgovor strežnika, preden je poslal naslednjo. Odstopanje je bilo izbrano tako, da so bili zadetki čim bolj enakomerno razporejeni po slikah v podatkovni bazi.

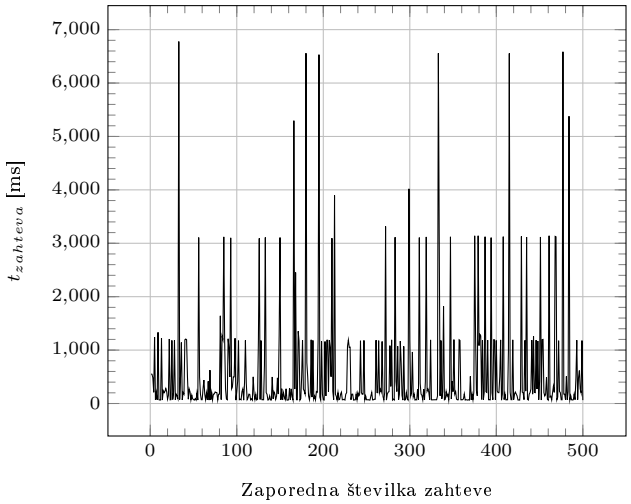
Zahteve je generiral osebni računalnik, nahajajoč se na Viču v Ljubljani. Prek brezžičnega omrežja je bil priklopljen na kabelski internet ponudnika A1, ki za zakupljen paket *A1 Kombo L* oglašuje hitrost povezave 40 megabitov na sekundo k uporabniku in 10 megabitov na sekundo od uporabnika. Test na strani *Speedtest by Ookla* te vrednosti potrdi (rezultat testa: `https://www.speedtest.net/result/8218379266`). Čas obhoda paketa od odjemalca do storitve in nazaj ($t_{mreza}$) je povprečju znašal 62 milisekund. Rezultat orodja `tracert` se nahaja v izpisu 1.3.

```
Tracing route to ec2-52-212-203-50.eu-west-1.compute.amazonaws.com
    [52.212.203.50]
over a maximum of 30 hops:

  1    2 ms   <1 ms   <1 ms 192.168.1.254
  2    *       *       *    Request timed out.
  3    *       *       *    Request timed out.
  4    *       *       *    Request timed out.
  5   25 ms   26 ms   25 ms 195.3.102.57
  6    *       *       *    Request timed out.
  7   26 ms   26 ms   28 ms lg4-9072.as8447.a1.net [195.3.64.142]
  8   26 ms   27 ms   25 ms 52.95.219.250
  9   31 ms   33 ms   32 ms 52.93.38.102
 10   25 ms   26 ms   27 ms 52.93.38.109
```

```
 11    *        58 ms    57 ms  54.239.44.47
 12   63 ms     58 ms    58 ms  52.93.128.141
 13   57 ms     60 ms    58 ms  52.93.128.2
 14    *        58 ms    63 ms  54.239.44.158
 15    *         *        *     Request timed out.
 16   72 ms     93 ms    80 ms  52.93.7.190
 17   59 ms     62 ms    59 ms  52.93.101.127
 18   72 ms      *       79 ms  52.93.101.126
 19   61 ms     61 ms    61 ms  52.93.36.143
 20    *         *        *     Request timed out.
 21    *         *        *     Request timed out.
 22    *         *        *     Request timed out.
 23    *         *        *     Request timed out.
 24    *         *        *     Request timed out.
 25    *         *        *     Request timed out.
 26    *         *        *     Request timed out.
 27   60 ms     62 ms    60 ms
     ec2-52-212-203-50.eu-west-1.compute.amazonaws.com [52.212.203.50]

Trace complete.
```

Listing 1.3: Rezultat orodja `tracert`.

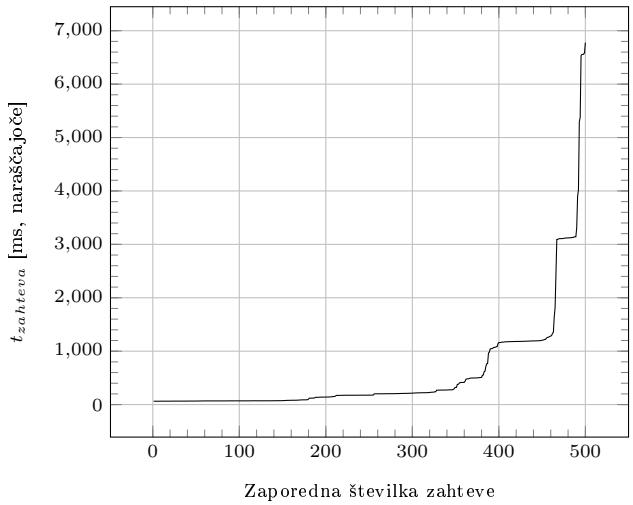
Celotni časi zahtev ($t_{zahteva}$) so prikazani na sliki 1.4.




Slika 1.4: Diagram celotnih časov zahtev.

Najhitrejši odziv $t_{zahteva}$ je znašal 63 milisekunde, najdaljši pa 6778 milisekund. V povprečju je čas odziva znašal 587 milisekund.

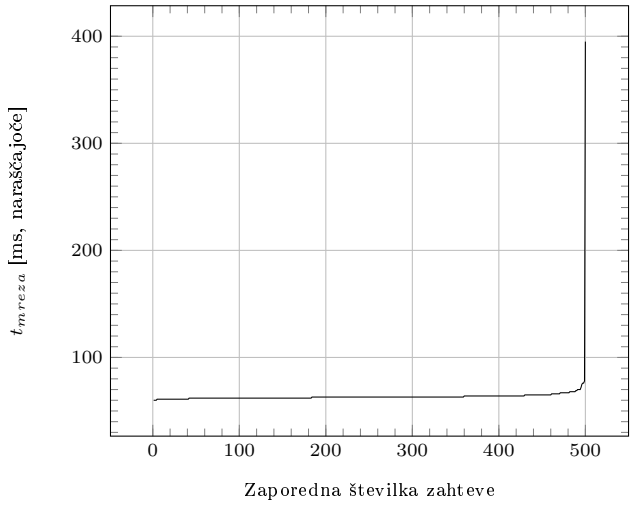
Iz slike 1.4 ni enostavno razvidno, kakšna je razporeditev časov zahtev. Na sliki 1.5 so celotni časi zahtev ($t_{zahteva}$) prikazani urejeni po velikosti.




Slika 1.5: Diagram celotnih časov zahtev, urejenih naraščajoče po velikosti.

Iz slike 1.5 je lažje razvidno, da je večina časov zahtev manj kot 500 milisekund, le peščica pa jih preseže 3500 milisekund.

Časi zahteve na mreži ($t_{mreza}$) so po velikosti urejeni prikazani na sliki 1.6. Najhitrejši čas obhoda je znašal 60 milisekund, najdaljši pa 395 milisekund. V povprečju je čas paketa na mreži znašal 64 milisekund, v 499 izmed 500 primerov ni presegel 78 milisekund.

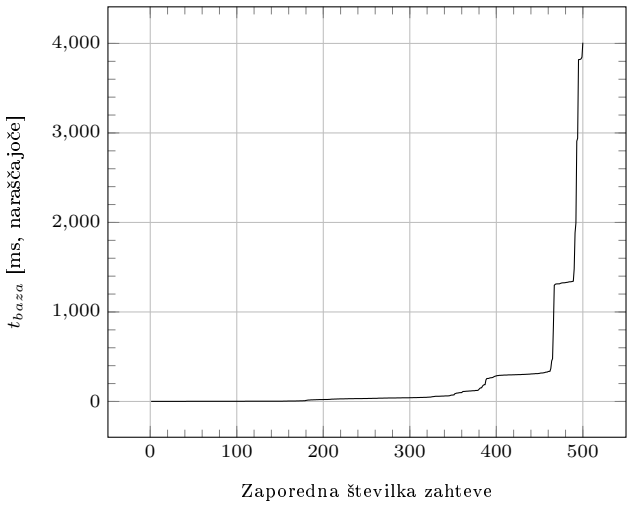



Slika 1.6: Diagram časov obhoda paketa, urejenih naraščajoče po velikosti.

V splošnem nas zanima, koliko časa vzame nalaganje slik z diska v pomnilnik ($t_{baza}$), kolikšen del procesiranja zavzame nalaganje slik ($\frac{t_{baza}}{t_{streznik}}$) in koliko časa

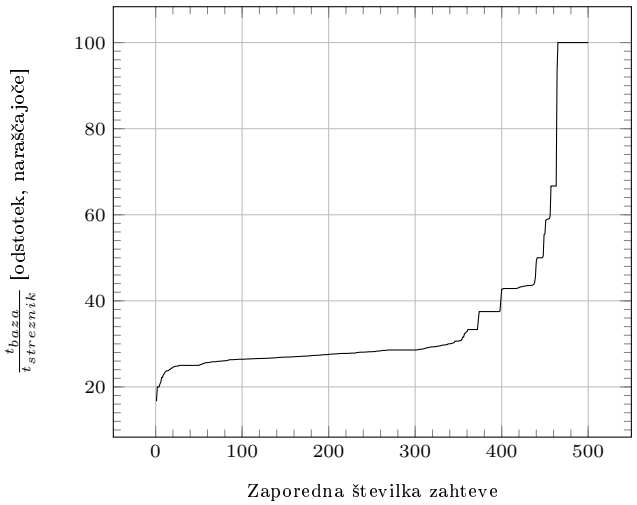
v povprečju zavzame nalaganje ene slike v pomnilnik ($\frac{t_{baza}}{N_{nalozenih}}$).

Časi, ki jih storitev porabi za nalaganje slik v pomnilnik, so po velikosti urejeni prikazani na sliki 1.7. Zavzemajo vrednosti med 1 in 4008 milisekund, v povprečju pa 201 milisekundo.



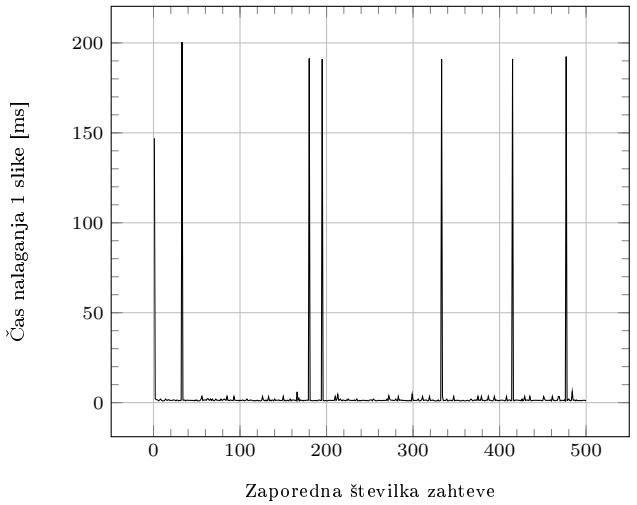

Slika 1.7: Diagram časov, porabljenih za nalaganje slik z diska v pomnilnik, urejenih naraščajoče po velikosti.

Odstotki časa, ki ga storitev porabi za nalaganje slik v pomnilnik, so po velikosti urejeni prikazani na sliki 1.8. Zavzemajo vrednosti med 16 in 100 odstotkov, v povprečju pa 36 odstotkov.




Slika 1.8: Diagram odstotkov časa, porabljenega za nalaganje slik z diska v pomnilnik, urejenih naraščajoče po velikosti.

Povprečne vrednosti časa nalaganja 1 slike so prikazane na sliki 1.9. Vrednosti so izračunane kot količnik časa nalaganja slik s številom preiskanih slik pri posamezni zahtevi. Zavzemajo vrednosti med 1 in 200 milisekund, v povprečju pa 4 milisekunde. V 493 izmed 500 primerov ni presegel 7 milisekund.




Slika 1.9: Diagram povprečnih časov nalaganja ene slike z diska v pomnilnik, v vrstnem redu zahtev.

### 1.4.2 Primerjava odzivov na zahteve iz različnih lokacij

Zaradi odločitve, da bomo testirali tudi z različnih lokacij, smo se najprej odločili, da preverimo kakšne zakasnitve (ping) ima posamezen član na različnih lokacijah do storitve. Rezultati po 30 ponovitvah so prikazani v tabeli 1.2. Te razlike v časih pripisujemo predvsem temu dejstvu, da ima vsaka od teh lokacij drugega ponudnika omrežnih storitev in zato naši paketi prepotujejo različne poti. V povprečju opravijo 7 skokov, dokler ne pridejo do skupne točke, ki je v lasti podjetja Amazon Technologies Inc. [10]. Te razlike so pomembne samo pri merjenju časa, ko se paket nahaja na mreži, in niso odvisne od zmogljivosti same storitve.

Tabela 1.2: Rezultat orodja `ping`

| Lokacija | Min [ms] | Max [ms] | Povprečje [ms] |
|---|---|---|---|
| Ljubljana Vič | 58 | 62 | 58 |
| Postojna | 41 | 41 | 41 |
| Vrtojba | 42 | 43 | 42 |

Testirali smo iskanje piksla, podobnega specifični barvi. Poslali smo 500 zahtev. Vsaka je vsebovala naključno barvo, strežnik pa je iskal piksel z odstopanjem, manjšim ali enakim 50. Zahteve so bile poslane zaporedno. Po oddaji zahteve je odjemalec počakal na odgovor strežnika, preden je poslal naslednjo. Minimalni, maksimalni in povprečni časi zahtev iz vsake lokacije so prikazani v tabeli 1.3.

Tabela 1.3: Časi zahtev $t_{zahteva}$, poslanih iz različnih lokacij.

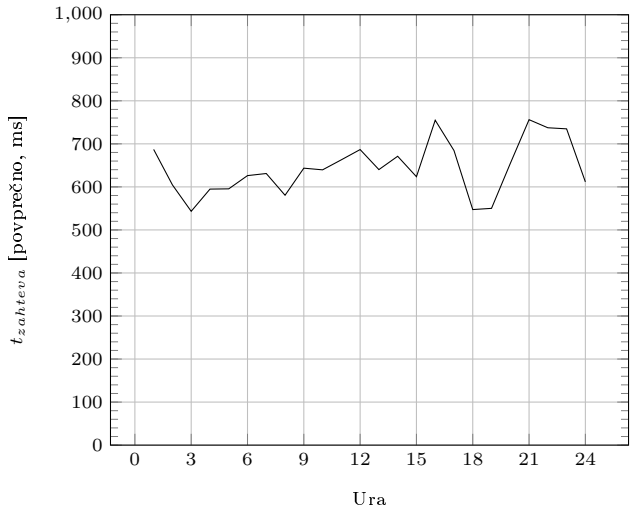
| Lokacija | Min [ms] | Max [ms] | Povprečje [ms] |
|---|---|---|---|
| Ljubljana Vič | 63 | 6778 | 587 |
| Postojna | 42 | 6850 | 624 |
| Vrtojba | 44 | 6570 | 549 |

Minimalni časi zahtev se z zanemarljivim odstopanjem ujemajo z izmerjenimi zakasnitvami do storitve. Variacijo pri maksimalnih in povprečnih časih zato pripisujemo nakjučni izbiri barve v zahtevah. Upoštevajoč to sklepamo, da ima lokacija odjemalca zanemarljiv pomen pri zmogljivosti storitve.

### 1.4.3 Primerjava odzivov na zahteve ob različnih časih

Kvaliteta storitve je odvisna tudi od zmožnosti odzivanja ob različnih časih. To smo testirali tako, da smo v urnih intervalih pošiljali 500 zahtev tipa iskanja piksla, podobnega specifični barvi. Vsaka zahteva je vsebovala naključno izbrano barvo, strežnik pa je iskal piksel z odstopanjem, manjšim ali enakim 50. Zahteve so bile poslane zaporedno, po oddaji zahteve pa je odjemalec počakal na odgovor

strežnika, preden je poslal naslednjo. Povprečni časi $t_{zahteva}$ glede na uro so prikazani na sliki 1.10.

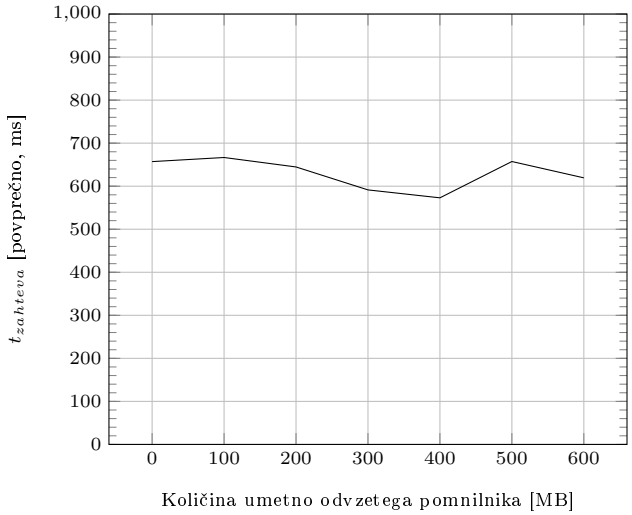



Slika 1.10: Diagram povprečnih časov odziva $t_{zahteva}$ glede na čas v dnevu.

V grobem se storitev odziva približno enako skozi dan, dopoldan in zgodaj zvečer nekoliko hitreje kot pozno popoldan in pozno zvečer.

### 1.4.4 Stresni test - primerjava odzivov na zahteve pri zmanjšani količini delovnega pomnilnika

Za izvedbo stresnega testa smo umetno zmanjševali količino delovnega pomnilnika, ki ga je storitev imela na voljo. To smo storili tako, da smo pred testiranjem zagnali program, ki je alociral poljubno količino delovnega pomnilnika in ga tako odvzel storitvi. Začeli smo s popolnoma prostim pomnilnikom (1 GB), nato pa ga postopoma alocirali v korakih po 100 MB. Ustavili smo se pri 600 MB alociranega pomnilnika, pri 700 MB je strežnik po nekaj zahtevah zmrznil in zahteval ponovni zagon programske opreme (vnos `Ctrl+C` v terminalu). Povprečni časi glede na količino umetno odvzetega pomnilnika so prikazani na sliki 1.11.




Slika 1.11: Diagram povprečnih časov odziva $t_{zahteva}$ glede na količino umetno odvzetega pomnilnika.

Opazili smo, da se storitev odziva približno enako, ne glede na količino razpoložljivega pomnilnika, v kolikor je na voljo vsaj približno 400 MB delovnega pomnilnika.

### 1.4.5 Bremenski test - primerjava odzivov na zahteve pri povečanemu število sočasnih uporabnikov

Za izvedbo bremenskega testa smo primerjali čase odzivov storitve pri enem uporabniku in čase odzivov pri treh sočasnih uporabnikih. Testirali smo iskanje piksla, podobnega specifični barvi. Za simulacijo treh sočasnih uporabnikov smo iz treh lokacij (Vič, Postojna in Vrtojba) hkrati zagnali iskanje. Časi so bili merjeni na Viču. Minimalni, maksimalni in povprečni časi zahtev enega in treh sočasnih uporabnikov so prikazani v tabeli 1.4.

Tabela 1.4: Časi zahtev $t_{zahteva}$ glede na število sočasnih uporabnikov.

| Število uporabnikov | Min [ms] | Max [ms] | Povprečje [ms] |
|---|---|---|---|
| 1 | 58 | 6551 | 662 |
| 3 | 58 | 23072 | 1960 |

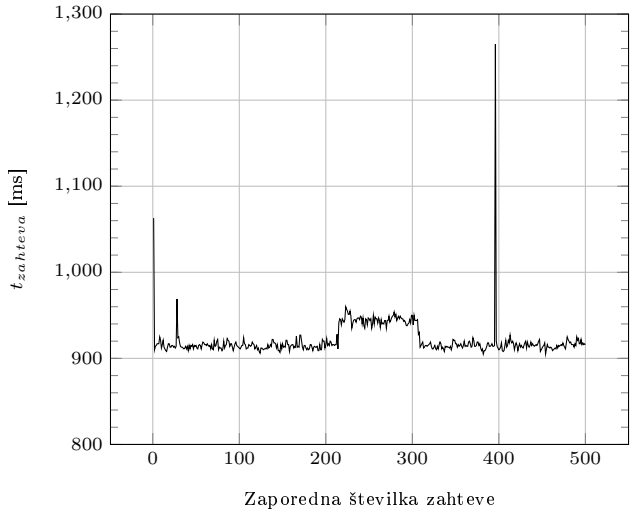
Minimalni čas je v obeh primerih enak. Je le čas, ki ga paket porabi od uporabnika do storitve in nazaj. Povprečni čas je v primeru treh uporabnikov približno trikrat daljši kot v primeru enega. To si lahko razlagamo na sledeči način: Storitev popolnoma zaposli že en uporabnik. Če imamo tri hkratne

uporabnike, mora potemtakem storitev vsakemu nameniti eno tretjino svoje računske moči. Povprečen čas odziva je zato trikrat daljši. Maksimalni čas je težje obrazložiti. V primeru treh uporabnikov je 3,5-krat daljši kot pri enem. Sklepamo, da smo naleteli na neznano ozko grlo, za natančnejšo določitev regresije pa bi morali natančneje opazovati obnašanje storitve pri več uporabnikih in določiti, kateri del obdelave povzroči to upočasnitev.

### 1.4.6 Frekvenčno testiranje

Zanima nas, kako se storitev odziva, če zahteve pošiljamo konstantno vsakih $\Delta t$ časovnih enot, $\Delta t$ pa po določenem času $t_{step}$ zmanjšamo in pri tem opazujemo, kako se podaljšuje čas odziva na zahtevo $t_{zahteva}$.

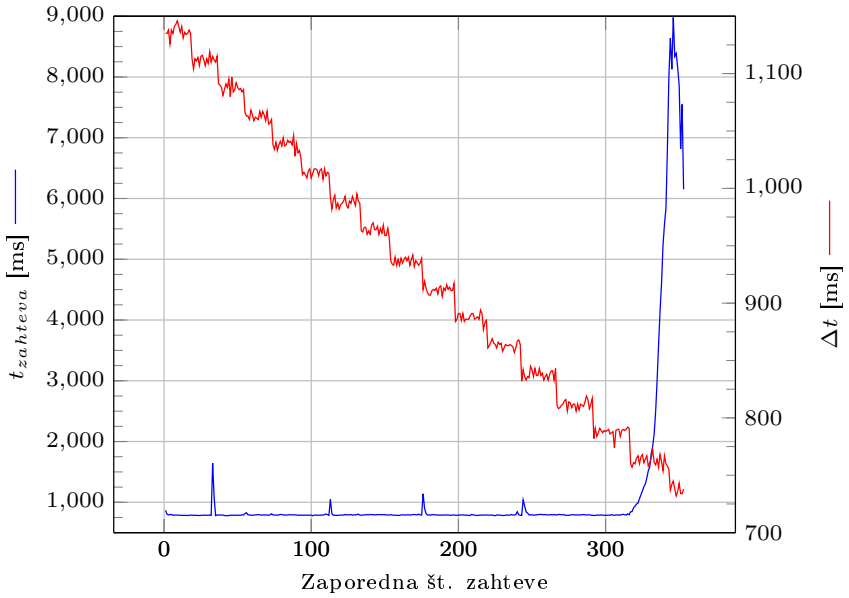
Za pravilno testiranje takega tipa smo se odločili, da bomo ves čas pošiljali isto zahtevo brez naključnosti na strežnik, kar pa lahko povzroči probleme, če imajo strežniki predpomnjenje. To smo stestirali tako, da smo poslali 500 enakih zahtev in opazovali čase $t_{zahteva}$, kar je prikazano na sliki 1.12.




Slika 1.12: Diagram časov odziva $t_{zahteva}$ pri istih zahtevah.

Če pogledamo graf, opazimo, da je skoraj ravna premica z izjemo nekaj špic. Če se posvetimo času $t_{processing}$, ki ni vrisan v graf, pa opazimo, da je med zahtevami zelo podoben, zato lahko te špice pripišemo zastojem na omrežju, ne pa času procesiranja. Posledično lahko razberemo, da pred našim strežnikom ni nobenega predpomnilnika, saj so naši časi procesiranja ostajali isti, časi zahtev pa so bili v določenih primerih višji, ne pa nižji, kot bi bili ob uporabi predpomnilnika. Hkrati ta naš zaključek potrjuje tudi dejstvo, da je vsaka od zahtev malo drugačna od druge, saj vsebuje svoj unikaten identifikator in uro pošiljanja, kar preprostemu predpomnilniku ne omogoča delovanja.

Za frekvenčno testiranje smo kot začetni $\Delta t$ izbrali vrednost 1,1 sekunde, za trajanje koraka $t_{step}$ smo izbrali vrednost 20 sekund. Po vsakem koraku smo $\Delta t$ zmanjšali za 25 milisekund. Dodatno smo vrednosti $\Delta t$ dodali še nekaj šuma z uporabo Poissonovega procesa, da smo dosegli boljšo simulacijo prihajanja zahtev v realnem svetu. Testiranje smo zaključili, ko je izmerjen čas odgovora presegel 8 sekund. Rezultati meritev so vidni na sliki 1.13




Slika 1.13: Graf frekvenčnega testiranja.

Iz grafa lahko razberemo, da se sistem zelo dobro odziva, dokler ni povprečni čas med dvema zahtevama manjši od 775 milisekund, to pomeni največ 1,29 zahteve na sekundo. Ko presežemo to mejo, začne $t_{zahteva}$ silovito naraščati, kar pomeni, da se začne ustvarjati čakalna vrsta za obdelavo, ki pa, kot vidimo, zelo hitro zrase. Zavajujoče deluje prevoj grafa $t_{zahteva}$, ki se začne spuščati, ko dosežemo mejo $\Delta t = 750ms$, kar pa pojasni dejstvo, da smo mi ravno pred tisto točko presegli zgornjo mejo $t_{zahteva} = 8000ms$ in po tem prenehali pošiljati zahteve na strežnik, vendar je strežnik še vedno moral sprazniti čakalno vrsto, kar je vidno v padcu grafa.

## 1.5 Zaključek

Platforma Amazon Elastic Cloud ponuja zanimivo brezplačno možnost za lansiranje enostavne spletne storitve. Sloni na razsežni oblačni infrastrukturi podjetja Amazon, kar zagotavlja hitre odzivne čase ne glede na lokacijo uporabnika in čas v dnevu, kar je razvidno iz naših testov. Ker so osnovne enote platforme

strežniki, na katerih lahko teče odprtokodna programska oprema, osnovana na jedru Linux, je snovanje in izvajanje programske opreme enostavno in dobro podprto. V sklopu merjenja brezplačne različice t2.micro z 1 jedrom in 1GB pomnilnika smo prišli do rezultatov, prikazanih v tabeli 1.5.

Tabela 1.5: Povzetek časov iz sheme opazovane storitve.

| Tip metrike (1 uporabnik) | Min [ms] | Max [ms] | Povprečje [ms] |
|---|---|---|---|
| $t_{zahteva}$ | 63 | 6778 | 587 |
| $t_{mreza}$ | 60 | 395 | 64 |
| $t_{streznik}$ | 3 | 6383 | 523 |
| $t_{baza}$ | 1 | 4008 | 201 |
| $t_{procesiranje}$ | 2 | 2375 | 322 |

Kot lahko vidimo, je največja težava, s katero se mora razvijalec spopasti, nizka zmogljivost brezplačnih resursov, ki so mu na voljo. Najbolj časovno zahtevno operacijo predstavlja procesiranje in nalaganje slikovnega materiala ($t_{streznik}$, ki združuje $t_{baza}$ in $t_{procesiranje}$), medtem ko je strošek komunikacije s strežnikom ($t_{mreza}$) karseda zanemarljiv. Že enostavna storitev lahko hitro zasede 1 jedro in 1 GB delovnega pomnilnika, poleg tega pa je treba pametno izkoristiti 30 GB navideznega diskovnega prostora, ki je na voljo brez plačila.

V kolikor nam omejena zmogljivost predstavlja težave, ima Amazon na voljo zmogljivejše produkte, ki pa so seveda plačljivi. Virtualni strežnik z dvemi jedri, 4 GB delovnega pomnilnika in 100 GB navideznega diskovnega prostora stane 23 evrov mesečno za najem [11], torej mora razvijalec šteti tudi profitabilnost med svoje cilje.

Amazon Elastic Cloud priporočamo za storitve, ki so računsko razmeroma nezahtevne. Uspeh storitev temelji na več faktorjih, vsekakor je zanesljivost infrastrukture zelo pomemben del. Tu Amazon Elastic Cloud zadosti vsaj osnovne potrebe, obljublja pa še veliko več, a ne brez plačila.

# Poglavje 2

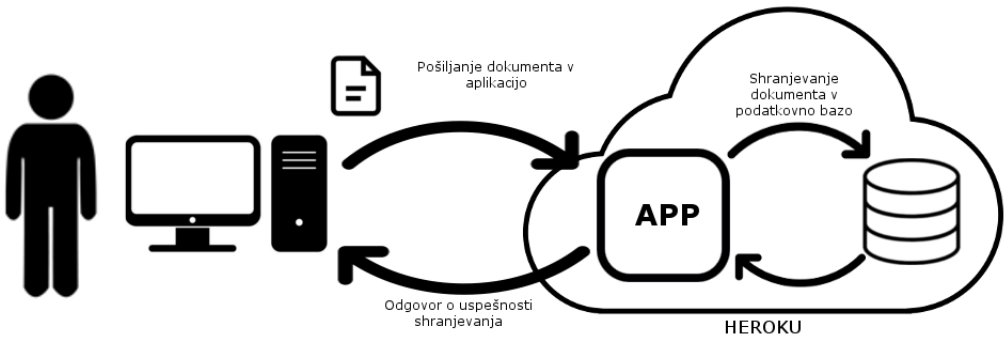
# Analiza zmogljivosti oblačne storitve Heroku

Jakob Gaberc Artenjak, Uroš Majcen, Ladislav Škufca

## 2.1 Opis storitve

Na trgu je na voljo veliko oblačnih storitev, ki nam ponujajo okolje za razvijanje in zaganjanje aplikacij. Ena izmed takih storitev je tudi Heroku. V našem projektu se bomo osredotočili predvsem na zmogljivost prenašanja datotek v in iz oblaka v tej platformi.

Zgradili bomo enostavno aplikacijo, ki bo omogočala prenašanje datotek v oblak, hrambo le teh na oblaku in prenašanje le teh iz oblaka. V praksi bo torej to izgledalo takole: uporabnik bo neko datoteko naložil v aplikacijo na oblaku, katera bo poskrbela, da se ta datoteka shrani v podatkovno bazo na oblaku. Po zaključku postopka bo uporabnik prejel obvestilo ali se je shranjevanje uspešno zaključilo ali ne. Grafičen prikaz uporabe storitve oblačne aplikacije je predstavljen na sliki 2.1.




Slika 2.1: Grafični prikaz opazovane oblačne storitve.

Podoben postopek bo potekal tudi v obratni smeri. Uporabnik bo izbral katere datoteke si želi pridobiti iz baze. Na podlagi izbranih datotek se bo poslala zahteva na strežnik, aplikacija pa bo po dobljeni zahtevi iz baze pridobila željene datoteke in jih poslala odjemalcu.

### 2.1.1 Platforma kot storitev

PaaS [12] (angl. *Platform as a Service*) je storitev, ki predstavlja obsežno porazdeljeno aplikacijsko okolje delovanja. Tako aplikacijsko okolje omogoča aplikaciji, da v celoti izkoristi procesne in pomnilniške vire v oblaku. Glavni namen takih storitev je olajšanje dela razvijalcem, saj lahko aplikacijo razvijejo, poženejo in upravljajo brez kompliciranega gradnje in vzdrževanja infrastrukture, ki je tipično potrebna za gradnjo in poganjanje aplikacije.

### 2.1.2 Heroku

Heroku [13] ponuja platformo kot storitev (PaaS). Razvijalci uporabljajo Heroku za postavitev in upravljanje modernih aplikacij. Podpira velik nabor programskih jezikov kot so Java, Python, PHP, Node.js, Scala, Ruby itd.

Pri naši implementaciji bomo uporabljali brezplačni plan Herukuja. Ta zajema 1000 brezplačnih "dyno" ur na mesec, aplikacijo pa samodejno pošlje v spanje po 30 minutah neuporabe (aplikacijo storitev postavi nazaj takoj, ko jo spet pokličemo preko URL). Brezplačni plan ponuja 512MB pomnilnika (RAM) in omogoča delo v skupinah do pet članov. Za bazo bomo uporabili Heroku Postgres, kar je Herokujeva implementacije PostgreSQL podatkovne baze. Le-ta brezplačno ponuja shranjevanje do 10.000 zapisov [14].

### 2.1.3 PostgreSQL

PostgreSQL [15] je odprtokodna objektno-relacijska podatkovna baza. Na voljo je tako na operacijskih sistemih Windows in Linux, kot tudi na macOS.

### 2.1.4 Node.js

Node.js [16] je platforma za razvoj programske opreme, kjer lahko kreiramo svoj lasten spletni strežnik in na njem zgradimo spletno aplikacijo. Vsebuje vgrajene knjižnice za izgradnjo HTTP strežnika. Pri razvijanju se uporablja jezik JavaScript.

V naši implementaciji bomo uporabili Express.js, ki je ogrodje za Node.js. Express.js je minimalno in robustno ogrodje za gradjo spletnih aplikacij, ki teži k optimalni izrabi virov pri delovanju. Ogrodje skrbi za čim manjše dodatno obremenjevanje performans delovanja. Pri testiranju v našem primeru si namreč želimo čim manj odvečnih stvari, ki bi lahko upočasnile delovanje.

## 2.2 Breme

Prenos podatkov med klientom in strežnikom lahko ponudnik storitve dodatno opremi z različnimi dodatki. Lahko doda "chunking" (rezanje večjih datotek na manjše dele, da ne preobremeni omrežja), "bundling" (kjer sestavi datoteko iz manjših datotek in jo nato pošlje na strežnik), deduplikacijo, "delta encoding" ali nek določen način kompresije [17].

Naša implementacija ne bo uporabljala nobene od naštetih strategij, pošiljali oz. prejemali in hranili bomo celotno datoteko. Datoteko bomo zakodirali na način `base64` [18], rezultat oz. kodni zapis pa shranili v podatkovno bazo. Kodiranje nam omogoča, da lahko v bazo na enak način shranimo poljuben tip datoteke. Z uporabo `base64` se velikost datoteke glede na nezakodirano velikost poveča za faktor 1.37 glede na vir [18] oz. 1.33 glede na vir [19] in lastne izkušnje.

### 2.2.1 Tipično datotečno breme

Ker pri pošiljanju in shranjevanju v naš oblačni strežnik obravnavamo datoteke enakovredno, med njimi ne bi smelo priti do razlik pod pogojem, da so enake velikosti in vse kodirane na način `base64`. Obremenitev na strežnik bo povsem enaka, dokler bo velikost datoteke enaka.

Našo storitev bomo testirali z datotekami različnimi velikosti. Odločili smo se za datoteke velikosti 10KB, 100KB in 1MB. Nekatere izmed teh velikosti predstavljajo tipične velikosti datotek, ki jih glede na izkušnje povprečni uporabnik prenese v oblačno storitev. Prav tako smo se za nekatere izmed teh velikosti odločili na podlagi pregleda poročil, ki so nastala pri tem predmetu v prejšnjih letih in druge literature [20]. Glede na to, da datoteke na klientu zakodiramo kot `base64`, nam je povsem vseeno, kateri tip datotek bomo uporabili pri pošiljanju na strežnik.

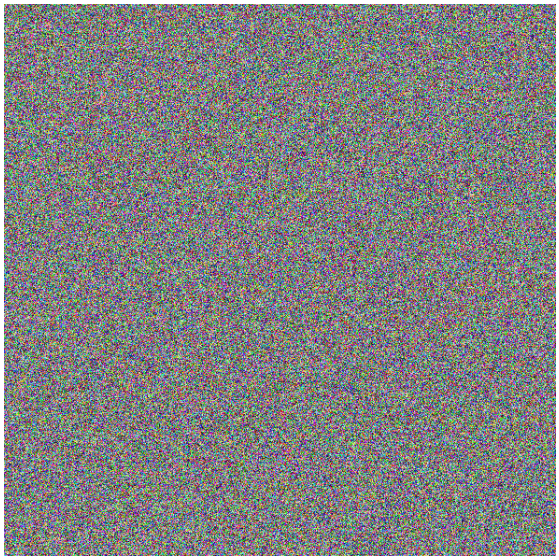
### 2.2.2 Generiranje datotek

Datoteke, katere bomo pošiljali na strežnik, želimo generirati tako, da so specifične velikosti in zvrsti ter se med seboj razlikujejo po vsebini. Za potrebe generiranja smo spisali `bash` skripto, ki nam generira poljubno število datotek

s poljubno vsebino (za poljuben tip in velikost). Glede na to, da smo pri realizaciji storitve opazili, da je na klientu izmed vseh tipov najlažje prikazati slike, smo za osnovni tip testne datoteke določili slike tipa .png. Izsek iz `bash` skripte, ki generira datoteke tipa .png, je viden v listingu 2.1. Generirana slika je zaradi načina delovanja skripte kvadratna, kar vodi v nenatančno velikost datoteke. To pomeni, da na primer velikost končne slike ni točno 1 MB, ampak je v takem primeru 1,1 MB. Pri izdelavi smo si pomagali z virom [21]. Primer generirane slike je viden na sliki 2.2.

```
size=($2*1024)/3
a=$(bc <<< "scale=0; sqrt($size)")

for i in $(eval echo {1..$1}); do
    mx=$a;
    my=$a;
    head -c "$((3*mx*my))" /dev/urandom | convert -depth 8 -size
        "${mx}x${my}" RGB:- pic$i.png
done
```

Listing 2.1: Izsek iz `bash` skripte, ki generira datoteke tipa .png.



Slika 2.2: Primer generirane slike velikost 1,1 MB tipa .png.

### 2.2.3 Čas oddajanja zahtev

Na začetku smo za čas oddajanja zahtev pri pošiljanju 10-ih datotek v oblak definirali vrednosti 1, 2, 3, 4 in 5 sekund. Te vrednosti smo določili povsem empirično, na podlagi dosedanjih izkušenj z oblačnimi storitvami. Odločili smo

se, da bomo pri izvajanju meritev čas oddajanja zahtev spreminjali na podlagi rezultatov, ki jih bomo pridobili na podlagi prejšnjih meritev.

## 2.3 Realizacija storitve

Kot že omenjeno je naša aplikacija zgrajena z ogrodjem Express.js. Glavna funkcionalnost aplikacije je shranjevanje uporabnikovih datotek v oblak. Uporabnik lahko v aplikaciji bodisi oddaja svoje datoteke v oblak ali jih iz njega prejema. Hkrati lahko preko preprostega uporabniškega vmesnika testira zmogljivost samega Herokuja.

Aplikacija na strani odjemalca za samo uporabo zahteva uporabniško prijavo za katero podatke (uporabniško ime in geslo) zgeneriramo ob postavitvi aplikacije. Po prijavi vidimo preprost uporabniški vmesnik, ki omogoča izbiro datotek (za pošiljanje na strežnik), določitev časa oddajanja zahtev, način procesa pošiljanja (enakomeren ali Poissonov) in polje za vnos čakalnega intervala. Čakalni interval predstavlja čas, ki ga odjemalec po končanem pošiljanju datotek počaka, predno zahteva vse rezultate pošiljanja (s tem preprečimo, da bi zahtevali izračun za datoteke, ki se mogoče še niso shranile v podatkovno bazo). Samo pošiljanje na strežnik izvedemo s klikom na gumb "Start upload". Na strani odjemalca se v rekurzivni zanki izvede funkcija za pošiljanje datoteke. Funkcija datoteko prebere z uporabo objekta FileReader, ki je vgrajen v JavaScriptu, njegove funkcije pa na dan pisanja popolnoma podpirata brskalnika Chrome in Edge, v katerih smo tudi testirali našo aplikacijo. FileReader potem vrne base64 zapis izbrane datoteke, kar z uporabo POST XMLHttpRequesta pošljemo na strežnik. Hkrati dodamo še ime datoteke in čas porajanja zahteve - $t_1$. Ker je aplikacija spisana v JavaScriptu in so vse funkcije privzeto asinhrone, na koncu zanke poskrbimo za sinhron klic naslednje datoteke glede na vnešen čas oddajanja. Na strežniku zabeležimo čas prispelega zahtevka - $t_2$, base64 zapis datoteke shranimo v podatkovno bazo, zabeležimo čas obdelave na strežniku - $t_3$ in zabeležimo čas, ko pride odgovor iz strežnika - $t_4$. Le-ta vsebuje tudi čase obdelave s strežnika, kar skupaj s časom prejetega odgovora shranimo v listo časov. Lista časov vsebuje podatke ($t_1$, $t_2$, $t_3$ in $t_4$) za vsako datoteko posebej. Ko se prenesejo vse datoteke, na strežnik v obdelavo pošljemo zahtevo s to listo časov. Strežnik na podlagi teh podatkov izračuna zahtevane čase, opisane v razdelku 2.4. Celoten postopek pošiljanja na strežnik lahko bolj podrobno spremljamo v razvijalski konzoli brskalnika (F12 - `developer tools`), kjer se izpisujejo podatki o trenutni akciji.

Po pošiljanju na strežnik vidimo dve tabeli - prva vsebuje čase za vsako datoteko posebej, druga pa vsebuje izračunane čase za cel test. V tabeli so vidni minimalni, maksimalni in povprečni časi. Poleg časov sta na voljo tudi minimalna, maksimalna in povprečna zasedenost RAM-a in CPU (več o tem v razdelku 2.4.2).

Aplikacija omogoča tudi izbris vseh datotek iz podatkovne baze s klikom na gumb "Delete DB entries".

V kolikor želimo pridobiti seznam vseh datotek, ga pridobimo s klikom na

gumb "Fetch files". Tu imamo tudi možnost ogleda posamezne datoteke v brskalniku (v kolikor je tip datoteke slika .png).

Že zgoraj omenjeni način procesa pošiljanja predstavlja način, kako se bodo datoteke pošiljale na strežnik. V kolikor je izbran enakomeren način, se bodo datoteke v rekurzivni zanki vedno pošiljale na strežnik v enakem časovnem intervalu, medtem ko se bodo v Poissonovem načinu pošiljale po Poissonovi porazdelitvi (več o tem v razdelku 2.5.4). Pri implementaciji tega načina smo uporabili knižnjico `poisson-process` [22].

Ob prvem testiranju implementirane aplikacije smo ugotovili, da je lahko čas oddajanja zahtev višji kot pa zmožnost hitrosti pasovne širine, ki jo dovoljuje ponudnik internetne storitve, zato datoteke v obdelavo na strežnik pridejo z zamikom, ki nastane zaradi omejitve pasovne širine. Prav tako se lahko zgodi, da ob zelo majhnem času oddajanja (cca. 30 ms) strežnik določene zahteve dobi prej, kot druge (na primer drugo poslano datoteko obdela prej kot prvo) - razlog se nahaja v tem, ker strežnik obdela prvo zahtevo, ki jo prejme, nekatere zahteve pa lahko zaradi omrežja pridejo prej kot druge.

### 2.3.1 Lokacije uporabe storitve

Uporabo storitve oz. izvajanje meritev smo opravljali na treh različnih lokacijah (hitrost internetne povezave smo izmerili z aplikacijo Speedtest[23]):

- Lokacija 1: Ljubljana Šiška, ISP Telekom, sistem z macOS 10.14.3 - `ping` 10 ms, hitrost prenašanja 9.99 Mbps, hitrost oddajanja 1.90 Mbps (čas merjenja 14.04.2019 ob 17:55),
- Lokacija 2: Ljubljana Rožna dolina - študentski domovi, ISP Arnes, sistem z Windows 10 - `ping` 0 ms, hitrost prenašanja 94.47 Mbps, hitrost oddajanja 92.44 Mbps (čas merjenja 14.04.2019 ob 11:04),
- Lokacija 3: Ljubljana Vič, ISP Telemach, sistem z Windows 10 - `ping` 9 ms, hitrost prenašanja 24.87 Mbps, hitrost oddajanja 5.85 Mbps (čas merjenja 16.04.2019 ob 09:48).

### 2.3.2 Objava na Heroku

Po objavi aplikacije na Heroku sta nas zanimala IP naslova strežnika in baze. IP naslov aplikacije smo pridobili s pomočjo klica Amazonove storitve za prikaz IP-ja [24], preverili pa še z IP Echo Service [25]. Obe storitvi smo z ukazom `curl` poklicali iz bash konzole na samem Heroku strežniku, pokazali pa sta IP 54.170.113.233 (18.04.2019 ob 21:00). Glede na objavo na Herokujevi strani s pomočjo [26] moramo upoštevati, da lahko Heroku ta IP kadarkoli spremeni. Gostiteljski naslov baze smo pridobili kar iz nadzorne plošče na Herokuju - `ec2-79-125-4-72.eu-west-1.compute.amazonaws.com` (18.04.2019 ob 21:21). V kolikor naredimo `ping` na ta naslov, dobimo IP 79.125.4.72. Že iz samega gostiteljskega naslova je razvidno, da se povezujemo na `amazonaws.com`. Izkaže se, da Heroku infrastrukturo gostuje pri Amazonu [27]. Kljub temu smo to

dejstvo preverili še z uporabo storitve IP Location Finder - Geolocation [28]. Storitev je pokazala, da se IP Heroku strežnika fizično nahaja v Dublinu na Irskem in je v lasti Amazon Technologies Inc. Enaka lokacija velja tudi za IP Postgresove baze.

Za obe lokaciji smo izvedli tudi `traceroute` in `ping` (za 10 paketov) iz Lokacije 1 dne 18.04.2019 okoli 22:00. `ping` na strežnik je prikazan v zapisu 2.2, `ping` na podatkovno bazo pa v zapisu 2.3. `traceroute` za strežnik je prikazan v zapisu 2.4, `traceroute` za podatkovno bazo pa v zapisu 2.5.

```
$ ping -c 10 54.170.113.233
PING 54.170.113.233 (54.170.113.233): 56 data bytes
64 bytes from 54.170.113.233: icmp_seq=0 ttl=45 time=47.724 ms
64 bytes from 54.170.113.233: icmp_seq=1 ttl=45 time=47.122 ms
64 bytes from 54.170.113.233: icmp_seq=2 ttl=45 time=46.392 ms
64 bytes from 54.170.113.233: icmp_seq=3 ttl=45 time=47.590 ms
64 bytes from 54.170.113.233: icmp_seq=4 ttl=45 time=49.386 ms
64 bytes from 54.170.113.233: icmp_seq=5 ttl=45 time=60.736 ms
64 bytes from 54.170.113.233: icmp_seq=6 ttl=45 time=47.286 ms
64 bytes from 54.170.113.233: icmp_seq=7 ttl=45 time=46.652 ms
64 bytes from 54.170.113.233: icmp_seq=8 ttl=45 time=48.062 ms
64 bytes from 54.170.113.233: icmp_seq=9 ttl=45 time=46.556 ms

--- 54.170.113.233 ping statistics ---
10 packets transmitted, 10 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 46.392/48.751/60.736/4.081 ms
```

Listing 2.2: `ping` na Heroku strežnik.

```
$ ping -c 10 ec2-79-125-4-72.eu-west-1.compute.amazonaws.com
PING ec2-79-125-4-72.eu-west-1.compute.amazonaws.com (79.125.4.72): 56
    data bytes
Request timeout for icmp_seq 0
Request timeout for icmp_seq 1
Request timeout for icmp_seq 2
Request timeout for icmp_seq 3
Request timeout for icmp_seq 4
Request timeout for icmp_seq 5
Request timeout for icmp_seq 6
Request timeout for icmp_seq 7
Request timeout for icmp_seq 8

--- ec2-79-125-4-72.eu-west-1.compute.amazonaws.com ping statistics ---
10 packets transmitted, 0 packets received, 100.0% packet loss
```

Listing 2.3: `ping` na podatkovno bazo.

```
$ traceroute 54.170.113.233
traceroute to 54.170.113.233 (54.170.113.233), 64 hops max, 52 byte
```

```
    packets
 1  gateway.home (192.168.1.1) 4.670 ms 5.762 ms 2.681 ms
 2  bsn-access.dynamic.siol.net (213.250.19.90) 12.824 ms 15.263 ms
     13.599 ms
 3  95.176.242.2 (95.176.242.2) 12.790 ms 12.196 ms 14.124 ms
 4  decix.amazon.com (80.81.194.152) 29.372 ms 27.549 ms 26.784 ms
 5  54.239.107.218 (54.239.107.218) 27.632 ms
    54.239.107.232 (54.239.107.232) 23.134 ms
    54.239.107.228 (54.239.107.228) 32.468 ms
 6  54.239.107.67 (54.239.107.67) 28.384 ms
    54.239.107.105 (54.239.107.105) 27.314 ms
    54.239.107.133 (54.239.107.133) 22.554 ms
 7  * * *
 8  176.32.106.248 (176.32.106.248) 52.578 ms 49.857 ms
    52.93.128.48 (52.93.128.48) 46.982 ms
 9  54.239.42.175 (54.239.42.175) 43.166 ms
    54.239.42.177 (54.239.42.177) 48.319 ms
    54.239.42.175 (54.239.42.175) 45.543 ms
10  * * *
11  52.93.6.156 (52.93.6.156) 44.806 ms
    52.93.6.170 (52.93.6.170) 45.994 ms
    52.93.6.136 (52.93.6.136) 71.417 ms
12  52.93.101.31 (52.93.101.31) 51.447 ms
    52.93.101.49 (52.93.101.49) 46.146 ms
    52.93.101.31 (52.93.101.31) 54.112 ms
13  52.93.101.48 (52.93.101.48) 59.548 ms
    52.93.101.42 (52.93.101.42) 49.905 ms
    52.93.101.30 (52.93.101.30) 49.239 ms
14  52.93.7.71 (52.93.7.71) 48.451 ms 50.447 ms
    52.93.7.69 (52.93.7.69) 49.397 ms
15  * * *
16  * * *
17  * * *
...
64  * * *
```

Listing 2.4: `traceroute` na Heroku strežnik.

```
$ traceroute ec2-79-125-4-72.eu-west-1.compute.amazonaws.com
traceroute to ec2-79-125-4-72.eu-west-1.compute.amazonaws.com
    (79.125.4.72), 64 hops max, 52 byte packets
 1  gateway.home (192.168.1.1) 2.572 ms 2.494 ms 2.342 ms
 2  bsn-access.dynamic.siol.net (213.250.19.90) 10.896 ms 13.389 ms
     12.288 ms
 3  95.176.242.2 (95.176.242.2) 11.734 ms 12.214 ms 12.400 ms
 4  decix.amazon.com (80.81.194.152) 26.431 ms 30.876 ms 28.870 ms
 5  54.239.107.230 (54.239.107.230) 49.299 ms
    54.239.107.220 (54.239.107.220) 22.875 ms
    54.239.107.232 (54.239.107.232) 22.425 ms
```

```
 6  54.239.107.45 (54.239.107.45) 26.928 ms
    54.239.107.17 (54.239.107.17) 27.189 ms
    54.239.107.175 (54.239.107.175) 24.855 ms
 7  * * *
 8  52.93.128.2 (52.93.128.2) 48.672 ms
    176.32.106.248 (176.32.106.248) 49.559 ms
    52.93.128.48 (52.93.128.48) 51.955 ms
 9  52.93.128.166 (52.93.128.166) 54.452 ms
    54.239.41.124 (54.239.41.124) 47.163 ms
    52.93.128.166 (52.93.128.166) 48.069 ms
10  * * *
11  52.93.6.210 (52.93.6.210) 76.962 ms
    52.93.6.156 (52.93.6.156) 70.803 ms *
12  52.93.101.41 (52.93.101.41) 49.573 ms
    52.93.101.55 (52.93.101.55) 67.791 ms
    52.93.101.59 (52.93.101.59) 43.614 ms
13  52.93.101.38 (52.93.101.38) 91.512 ms
    52.93.101.52 (52.93.101.52) 48.220 ms
    52.93.101.0 (52.93.101.0) 50.002 ms
14  52.93.7.133 (52.93.7.133) 47.729 ms
    52.93.7.149 (52.93.7.149) 55.424 ms
    52.93.7.155 (52.93.7.155) 49.238 ms
15  * * *
16  * * *
17  * * *
...
64  * * *
```

Listing 2.5: `traceroute` na podatkovno bazo.

Ugotovili smo, da `ping` nikakor ne pride do IP-ja baze (ne glede na to, ali pingamo direktno na IP naslov ali naslov gostitelja), `ping` na strežnik pa nam nazorno pokaže povprečni čas 46,392 ms.

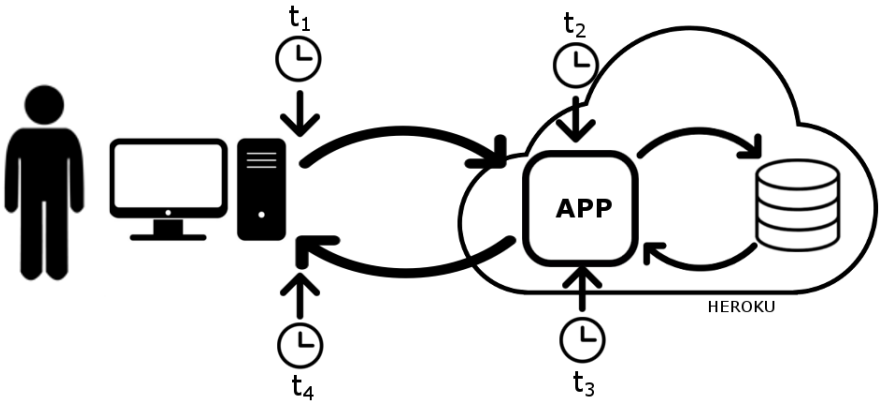
Traceroute strežnika se konča pri naslovu 52.93.7.69, traceroute baze pa na naslovu 52.93.7.155, ki se glede na IP Location Finder - Geolocation [28] nahajata v Washingtonu v Ameriki, katere lastnik je prav tako Amazon Technologies Inc. Razloga, zakaj se `traceroute` preusmeri v Ameriko in tam ustavi, nismo našli.

S tem ko smo aplikacijo objavili na Heroku, smo pred izvajanjem testov želeli vedeti, kako se le-ta obnaša v Herokujevem okolju. Nekajkrat smo poslali različno število datotek velikost 1,1 MB z različnimi časi oddajanja in ugotovili, da le-ta deluje tako, kot smo si zamislili. Prav tako smo nekajkrat preverili tudi prikaz slik v samem brskalniku. Zataknilo pa se je pri akciji "Fetch files", ko smo želeli pridobiti seznam vseh datotek. Če smo naložili 100 slik tipa .png, pri čemer je bila vsaka datoteka velikosti 1,1 MB, se je ob akciji "Fetch files" sesula celotna aplikacija. Problem je nastal zaradi omejitve pomnilnika na strani Herokuja (500 MB). Ko pokličemo seznam vseh datotek, si le-te strežnik shrani v RAM-u in potem zahtevo posreduje nazaj na odjemalca. V kolikor pridobimo

100 slik velikost 1,1 MB, to znaša najmanj 110 MB. Temu moramo prišteti tudi ostale meta informacije in pa zasedenost pomnilnika s strani same aplikacija. Skupni seštevek je presegel mejo 500 MB, zaradi česar je Heroku aplikacijo zaustavil. Po nekaj poizkusih smo ugotovili, da ponovno postavitev aplikacije izvede Heroku sam (po nekaj minutah), ali pa aplikacijo ponovno postavimo tako, da izvedemo ponovno namestitev (ang. `deploy`).

## 2.4 Metrike

Kot že opisano v prejšnjem poglavju bomo pri prenašanju datotek merili čase na štirih lokacijah, ki so predstavljeni na sliki 2.3.




Slika 2.3: Grafični prikaz nalaganja datoteke na oblak.

### 2.4.1 Pošiljanje datotek v oblak

Čas $t_1$ predstavlja trenutek, ko se sproži POST zahteva za pošiljanje datoteke na strežnik. Čas $t_2$ predstavlja trenutek, ko strežnik to datoteko sprejme in začne s shranjevanjem v podatkovno bazo. Takoj po shranjevanju teh datotek v bazo zabeležimo čas $t_3$. Ko na odjemalcu dobimo odgovor iz strežnika, zabeležimo še čas $t_4$. Iz zabeleženih časov lahko določimo celotni čas našega testa $T_{total} = t_4 - t_1$, čas obdelave na strežniku $T_{service} = t_3 - t_2$ in omrežni čas $T_{network} = T_{total} - T_{service}$.

Izračunali smo tudi čas nalaganja v strežnik ($T_{upload} = t_2 - t_1$) in prejemanja iz strežnika ($T_{download} = t_4 - t_3$). Ker sta časa $t_1$ in $t_4$ merjena na odjemalcu, časa $t_2$ in $t_3$ pa na strežniku, je prišlo do zanimivih rezultatov. V kolikor smo aplikacijo uporabljali na sistemu Windows 10, je bil čas $T_{upload}$ negativen, v kolikor pa smo aplikacijo uporaljali na sistemu macOS 10.14.3, je bil čas $T_{upload}$ pozitiven. Po natančnem pregledu same aplikacije smo ugotovili, da mora do negativnih časov pri uporabi sistema Windows 10 prihajati zaradi različne sinronizacije sistemske ure. Ker je strežnik nameščen na operacijskem sistemu

Ubuntu 18.04.2 LTS, očitno s sistemom macOS uporablja isti strežnik za sinhronizacijo ure, pri sistemu Windows 10 pa temu ni tako. Kljub temu, da smo Windows 10 prisilili, da uporablja enak strežnik za sinhronizacijo ure kot sistem Ubuntu, so bile razlike v času kljub vsemu negativne. Ker do zamika ure pride tako pri izračunu časa $T_{download}$ kot tudi pri izračunu časa $T_{upload}$, je celotni izračun $T_{network}$ še vedno pravilen.

### 2.4.2 Strežniške metrike

Ker nas je poleg samih časov zanimalo tudi stanje pomnilnika in CPU na strežniku, smo v aplikacijo implementirali orodje js-meter [29]. Gre za preprosto orodje, ki sporoča stanje RAM-a in čas CPU-ja pri obdelavi neke akcije. Po pregledu strežniških zapisov (ang. `server logs`) smo ugotovili, da so nekatere akcije precej pomnilniško zahtevne (na primer branje vseh trenutno shranjenih datotek v podatkovni bazi - "Fetch files"). Hkrati smo s nekaj preprostimi testi preverili oz. potrdili verodostojnost izmerjenih časov na strežniku. Izkaže se, da orodje pri obdelavi in shranjevanju datoteke na strežnik sporoči isti čas obdelave, kot smo ga izračunali sami.

Merili smo tudi stanje oz. zasedenost CPU v %, vendar smo se odločili, da ga ne bomo podali poleg rezultatov meritev. Izkaže se, da je poraba v % podana za vsa virtualna jedra na Herokuju, ki pa se spreminjajo "on the fly" - glede na trenutno porabo.

Poleg orodja js-meter smo na strani Herokuja omogočili funkcijo `log-runtime-metrics`. Gre za funkcijo iz sklopa Herokujevega Heroku Labs [30]. Heroku Labs je skupek orodij oz. funkcij, ki so v pripravi oz. razvoju in se preučujejo, da bi bile dodane v Heroku. `log-runtime-metrics` nam omogočajo detajlni vpogled v samo stanje kontejnerja, ki nam je v Heroku na voljo (v primerjavi z js-meter torej nismo vezani na aplikacijo). Za konkreten pogled nad omogočenimi metrikami smo uporabili Herokujev dodatek (ang. `add-on`) Librato [31]. V osnovi je orodje zastonj, omogoča pa nam vizualni pregled metrik za 1 uro nazaj z osveževalnim časom 60-ih sekund.

Za potrebe razumevanja teh metrik smo preučili upravljanje s pomnilnikom v ogrodju Node.js.

## 2.5 Rezultati in meritve

V tem razdelku predstavimo naše meritve in teste, ki smo jih izvedli. Poleg vsakega testa so podani tudi rezultati in končne ugotovitve za posamezen test.

### 2.5.1 Meritve

Pred začetkom meritev oz. izvajanjem določenega testa smo vedno pobrisali vse podatke iz podatkovne baze, da je bilo izhodišče za vse teste vedno enako. Prav tako smo na sistemu, ki je upravljal z odjemalcem, vedno zaprli vse aplikacije

(odprte s strani uporabnika operacijskega sistema), ki bi lahko vplivale na čas $T_{network}$.

### 2.5.2 Prvi test - pošiljanje 10 slik tipa .png

Pri prvem testu smo na strežnik 5-krat pošiljali 10 slik tipa .png, velikosti 1,1 MB z različno izbranim časom oddajanja (vsakič isti nabor 10-ih slik v istem zaporedju, slike pa imajo med seboj različno vsebino - generirano naključno). Za testiranje je bila uporabljena lokacija 3.

#### Hipoteza

Predpostavljamo, da velikih razlik pri času $T_{service}$ ne bo, saj shranjujemo zgolj 10 slik.

#### Rezultati

Test 1-1: 10 slik iste velikosti z naključno generirano vsebino. Čas oddajanja datotek 5 sekund. Lokacija 3, 07.04.2019 ob 19:50. Rezultati testa so vidni v tabeli 2.1.

| | $T_{total}$ | $T_{service}$ | $T_{network}$ |
|---|---|---|---|
| avg | 3558.50 ms | 186.00 ms | 3372.50 ms |
| max | 4771.00 ms | 318.00 ms | 4540.00 ms |
| min | 2844.00 ms | 123.00 ms | 2681.00 ms |

Tabela 2.1: Rezultati testa 1-1.

Test 1-2: 10 slik iste velikosti z naključno generirano vsebino. Čas oddajanja datotek 4 sekunde. Lokacija 3, 07.04.2019 ob 19:55. Rezultati testa so vidni v tabeli 2.2.

| | $T_{total}$ | $T_{service}$ | $T_{network}$ |
|---|---|---|---|
| avg | 3559.00 ms | 200.20 ms | 3358.80 ms |
| max | 4901.00 ms | 279.00 ms | 4776.00 ms |
| min | 2909.00 ms | 131.00 ms | 2721.00 ms |

Tabela 2.2: Rezultati testa 1-2.

Test 1-3: 10 slik iste velikosti z naključno generirano vsebino. Čas oddajanja datotek 3 sekunde. Lokacija 3, 07.04.2019 ob 19:58. Rezultati testa so vidni v tabeli 2.3.

| | $T_{total}$ | $T_{service}$ | $T_{network}$ |
|---|---|---|---|
| avg | 4265.40 ms | 153.40 ms | 4112.00 ms |
| max | 7889.00 ms | 199.00 ms | 7739.00 ms |
| min | 2940.00 ms | 127.00 ms | 2805.00 ms |

Tabela 2.3: Rezultati testa 1-3.

Test 1-4: 10 slik iste velikosti z naključno generirano vsebino. Čas oddajanja datotek 2 sekunde. Lokacija 3, 07.04.2019 ob 20:00. Rezultati testa so vidni v tabeli 2.4.

| | $T_{total}$ | $T_{service}$ | $T_{network}$ |
|---|---|---|---|
| avg | 7323.00 ms | 293.80 ms | 7029.10 ms |
| max | 12476.00 ms | 654.00 ms | 12322.00 ms |
| min | 3905.00 ms | 154.00 ms | 3692.00 ms |

Tabela 2.4: Rezultati testa 1-4.

Test 1-5: 10 slik iste velikosti z naključno generirano vsebino. Čas oddajanja datotek 1 sekunda. Lokacija 3, 07.04.2019 ob 20:02. Rezultati testa so vidni v tabeli 2.5.

| | $T_{total}$ | $T_{service}$ | $T_{network}$ |
|---|---|---|---|
| avg | 11970.70 ms | 440.40 ms | 11530.30 ms |
| max | 18312.00 ms | 1269.00 ms | 17668.00 ms |
| min | 4913.00 ms | 172.00 ms | 4702.00 ms |

Tabela 2.5: Rezultati testa 1-5.

**Ugotovitve**

Pri pošiljanju 10 datotek se je izkazalo, da 4 sekunda razlika (glede na rezultate v tabeli 2.1 in 2.5) v času med generiranjem zahtev pošiljanja poveča povprečni čas obdelave na strežniku ($T_{service}$) za 254,4 ms. Na podlagi tega eksperimenta moramo našo hipotezo zavreči.

### 2.5.3 Drugi test - pošiljanje 100 slik tipa .png

Pri drugem testu smo na strežnik 3-krat pošiljali 100 slik tipa .png, velikosti 1,1 MB z različno izbranim časom oddajanja (vsakič isti nabor 100-ih slik v istem zaporedju, slike pa imajo med seboj različno vsebino - generirano naključno). Najprej smo slike pošiljali na 5 sekund, kjer smo opazili, da povprečni skupni čas ($T_{total}$) znaša 3780.90 ms, kar je vidno v tabeli 2.6. Iz tega razloga smo naslednji test ponovili s časom oddajanja 3781 ms, zadnji test pa z 1000 ms. Za testiranje je bila uporabljena lokacija 3.

### Hipoteza

Predpostavljamo, da bo prišlo do velikih razlik pri času $T_{service}$, saj shranjujemo 100 slik. Pričakovana razlika v času vsaj 300 ms.

### Rezultati

Test 2-1: 100 slik iste velikosti z naključno generirano vsebino. Čas oddajanja datotek 5 sekund. Lokacija 3, 07.04.2019 ob 20:07. Rezultati testa so vidni v tabeli 2.6.

| | $T_{total}$ | $T_{service}$ | $T_{network}$ |
|---|---|---|---|
| avg | 3780.90 ms | 144.60 ms | 3636.30 ms |
| max | 13719.00 ms | 281.00 ms | 13575.00 ms |
| min | 2542.00 ms | 110.00 ms | 2400.00 ms |

Tabela 2.6: Rezultati testa 2-1.

Test 2-2: 100 slik iste velikosti z naključno generirano vsebino. Čas oddajanja datotek 3781 milisekund. Lokacija 3, 07.04.2019 ob 20:38. Rezultati testa so vidni v tabeli 2.7.

| | $T_{total}$ | $T_{service}$ | $T_{network}$ |
|---|---|---|---|
| avg | 3789.70 ms | 148.60 ms | 3641.10 ms |
| max | 9587.00 ms | 594.00 ms | 9452.00 ms |
| min | 2670.00 ms | 109.00 ms | 2552.00 ms |

Tabela 2.7: Rezultati testa 2-2.

Test 2-3: 100 slik iste velikosti z naključno generirano vsebino. Čas oddajanja datotek 1 sekunda. Lokacija 3, 07.04.2019 ob 20:20. Rezultati testa so vidni v tabeli 2.8.

| | $T_{total}$ | $T_{service}$ | $T_{network}$ |
|---|---|---|---|
| avg | 84405.80 ms | 252.10 ms | 84153.70 ms |
| max | 146569.00 ms | 1598.00 ms | 146039.00 ms |
| min | 7318.00 ms | 114.00 ms | 7117.00 ms |

Tabela 2.8: Rezultati testa 2-3.

### Ugotovitve

Razlika v povečanju časa obdelave na strežniku ($T_{service}$) je precej manj opazna kot pri testu 1, se pa tu začnejo pojavljati motnje v omrežju - sprejemanje datotek na strežniku ni več zaporedno, kar vodi v kolaps (povprečni čas za pot ene datoteke naraste iz 3,8 s (pri času oddajanja 5 sekund) na kar 84 sekund

(pri času oddajanja 1 sekunda). Izkaže se, da se čas obdelave $T_{service}$ ni tako velik, kot smo pričakovali - našo hipotezo ponovno zavrnemo.

### 2.5.4 Tretji test - pošiljanje 100 slik tipa .png z uporabo Poissonove porazdelitve

V realnosti se vsako pošiljanje na strežnik izvaja z nekim različnim časovnim intervalom (in ne fiksnim, tako kot v naših testih). Zaradi tega smo v naši aplikaciji implementirali možnost pošiljanja datotek po Poissonovi porazdelitvi [32], ki predstavlja bolj realen scenarij. S tem testom smo preverili, kako opazna je razlika v časih.

Pošiljanje po Poissonovi porazdelitvi smo implementirali s pomočjo knjižnice [22]. Vzorec pridobimo s klicem funkcije `poissonProcess.sample(X)`, pri čemer X v našem primeru predstavlja čas oddajanja v milisekundah. Knjižnica implementira porazdelitveno funkcijo, prikazano na sliki 2.4 [33]. Na podlagi vnešenega časa X nam knjižnica vrne vzorec iz omenjene porazdelitve s stopnjo 1/X. Za več informacij o samem procesu v branje predlagamo dokumentacijo uporabljene knjižnice [22].

$$F(x;\lambda) = \begin{cases} 1 - e^{-\lambda x} & x \geq 0, \\ 0 & x < 0. \end{cases}$$

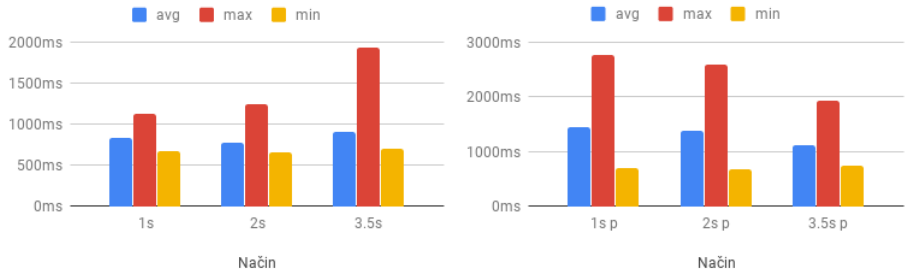
Slika 2.4: Porazdelitvena funkcija za diskretno spremenljivko X po Poissonovi porazdelitvi [33].

Na strežnik smo iz lokacije 2 v soboto 20.04.2019 (okoli 8:00) pošiljali 100 slik tipa .png, velikosti 1,1 MB (vsakič isti nabor 100-ih slik v istem zaporedju, slike pa imajo med seboj različno vsebino - generirano naključno). Čas oddajanja je bil definiran kot 1s, 2s in 3.5s (glede na rezultate prvega testa), pri čemer je bil test za vsako čas oddajanja izveden dvakrat - enkrat s enakomernim intervalom oddajanja za vsako datoteko, drugič z vključeno Poissonovo porazdelitvijo. Pri tem testu smo prvič merili tudi porabo RAM-a na strežniku.

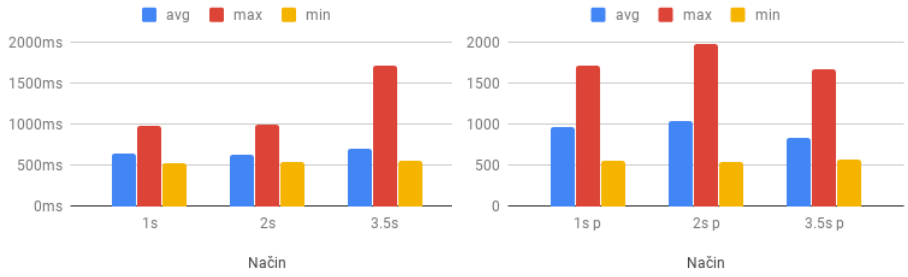
#### Hipoteza

Predpostavljamo, da bo prišlo do spremembe $T_{total}$ in $T_{network}$ zaradi drugačnega časa oddajanja, pri čemer se $T_{service}$ ne bo spreminjal.

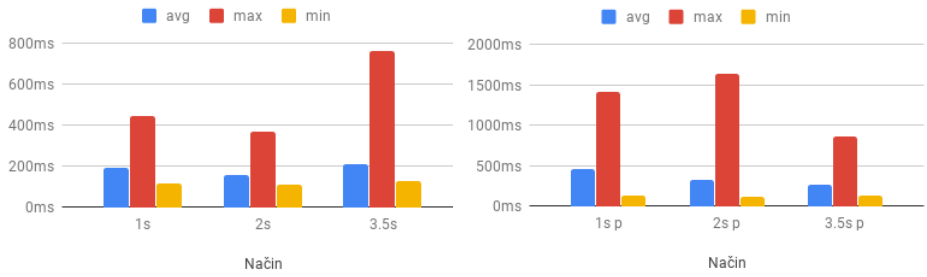
**Rezultati**




Slika 2.5: Rezultati $T_{total}$ za testiranje z različno porazdeljenim časom oddajanja - levo je čas oddajanja enakomeren, desno je porazdeljen po Poissonovi porazdelitvi.




Slika 2.6: Rezultati $T_{network}$ za testiranje z različno porazdeljenim časom oddajanja - levo je čas oddajanja enakomeren, desno je porazdeljen po Poissonovi porazdelitvi.




Slika 2.7: Rezultati $T_{service}$ za testiranje z različno porazdeljenim časom oddajanja - levo je čas oddajanja enakomeren, desno je porazdeljen po Poissonovi porazdelitvi.

#### Ugotovitve

Izkaže se, da smo se z našo hipotezo močno zmotili. Ob prvem pogledu na sliko 2.5 bi res lahko rekli, da naša hipoteza drži, toda ob bolj detajlnem pregledu slike 2.7 vidimo, da ni zgolj $T_{network}$ kriv za porast $T_{total}$. Kljub temu na sliki 2.6 lepo vidimo višji porast $T_{network}$ pri časih oddajanja 1 in 2 sekundi.

Glede na rezultate menimo, da Poissonova porazdelitev poskrbi, da se določene slike odpošljejo tako hitro, da na strežnik priletijo ob praktično istem času (pri časih oddajanja 1 in 2 sekundi). Zaradi tega se poveča čas $T_{service}$ in pa tudi $T_{network}$. Zanimivo je, da vpliv Poissona praktično ni več očiten pri času oddajanja 3.5 sekunde. Poissonova porazdelitev ima torej občuten vpliv pri manjših časih oddajanja.

Povprečna zasedenost polnilnika RAM na strežniku je pri vseh 6-ih meritvah znašala 217 MB in se ni niti povišala niti zmanjšala glede na način pošiljanja za razliko 10%.

### 2.5.5 Četrti test - 20-urno pošiljanje 100 slik tipa .png

Pri četrtem testu smo na strežnik vsako uro v obdobju 20 ur pošiljali 100 slik tipa .png, velikosti 1,1 MB (vsakič isti nabor 100-ih slik v istem zaporedju, slike pa imajo med seboj različno vsebino - generirano naključno). Čas oddajanja zahtev je bil definiran kot 1s, 2s in 3.5s (glede na rezultate prvega testa). Zaradi rezultatov tretjega testa, so bili vsi časi oddajanja vzorčeni po Poissonovi porazdelitvi. S testom smo pričeli v petek 19.04 ob 12:00, zaključili pa v soboto 20.04.2019 ob 7:00 (meritve so se izvajale vsako polno uro). Kot vemo, ob večerih promet po omrežju naraste, kar bi lahko vplivalo na naše meritve. To smo počeli na lokaciji 2, saj ima le-ta najboljšo internetno povezavo.

#### Hipoteza

Predpostavljamo, da bo v poznejših urah prišlo do večje obremenitve omrežja, s čimer se nam bodo povišali časi $T_{total}$, $T_{service}$ in $T_{network}$.

**Rezultati**

Test 4-1: 100 slik iste velikosti z naključno generirano vsebino. Čas oddajanja datotek 3.5 sekunde po Poissonu. Lokacija 2, 19.04.2019 ob 12:11. Rezultati testa so vidni v tabeli 2.9.

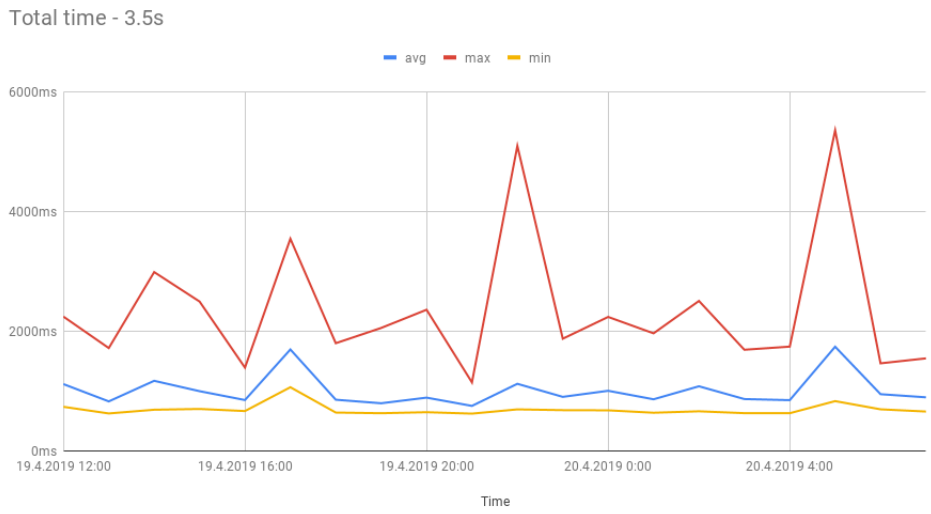
| | $T_{total}$ | $T_{service}$ | $T_{network}$ | $T_{download}$ | $T_{upload}$ | RAM |
|---|---|---|---|---|---|---|
| avg | 1123.50 ms | 330.85 ms | 792.65 ms | -1728.49 ms | 2521.14 ms | 236.82 MB |
| max | 2251.00 ms | 1422.00 ms | 1411.00 ms | 0.00 ms | 3144.00 ms | 270.70 MB |
| min | 744.00 ms | 141.00 ms | 562.00 ms | -1736.00 ms | 2294.00 ms | 189.35 MB |

Tabela 2.9: Rezultati testa 4-1.

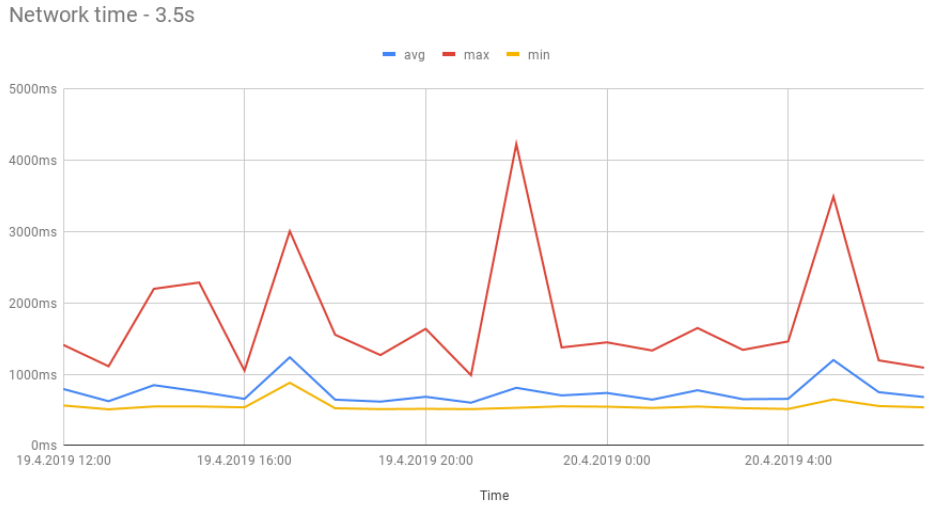
Na podlagi zgornje tabele vidimo problem, ki smo ga opisali v poglavju 2.4.1. Zaradi omenjega problema smo v naslednjih testih opustili zapise časov $T_{download}$ in $T_{upload}$.

Ker je rezultatov preveč za lepo razviden tabelaričen prikaz, smo se v tem primeru odločili za grafičnega. Prav tako je preveč vseh možnih grafov, zato smo se na koncu odločili, da prikažemo grafe za testiranje s časom oddajanja 3.5 s, saj so se nam ti po pregledu rezultatov meritev zdeli najbolj zanimivi.

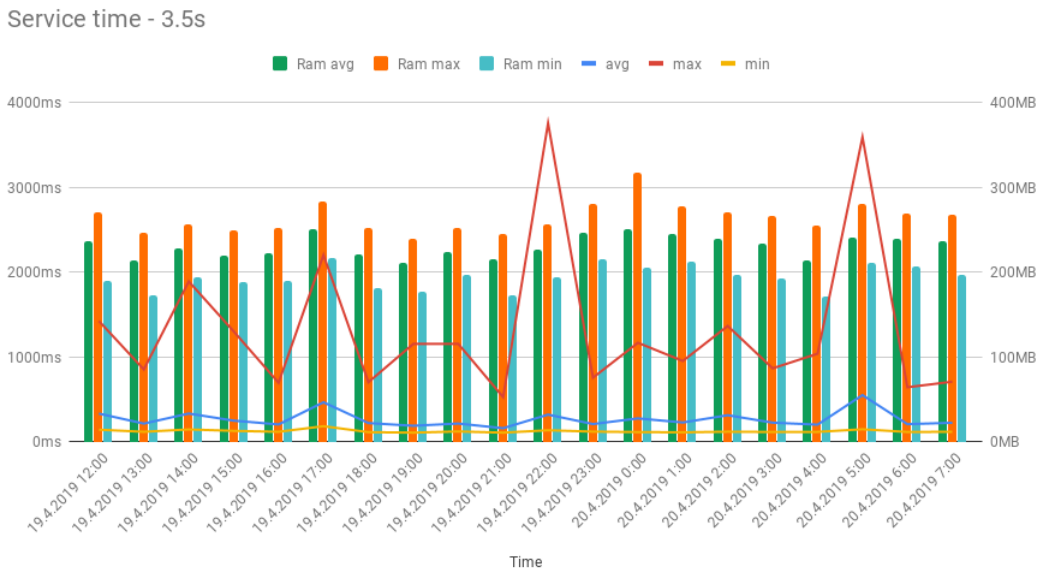
Na sliki 2.8 je prikazan graf za $T_{total}$, na sliki 2.9 je prikazan graf za $T_{network}$ in na sliki 2.10 je prikazan graf za $T_{service}$. Ta graf vsebuje tudi podatke o porabi pomnilnika na Heroku strežniku.




Slika 2.8: Rezultati $T_{total}$ za testiranje s časom oddajanja 3.5 s.




Slika 2.9: Rezultati $T_{network}$ za testiranje s časom oddajanja 3.5 s.




Slika 2.10: Rezultati $T_{service}$ in porabe RAM-a za testiranje s časom oddajanja 3.5 s.

#### Ugotovitve

Pri izvajanju tega testa smo bili še posebej pozorni na vključene sistemske metrike. Opazili smo, da je bila največja obremenitev pomnilnika pri meritvah okoli polnoči, kljub temu pa je količina porabljenega pomnilnika skozi vse teste znašala okoli 250MB, kar je okoli 50% vsega pomnilnika, ki nam je na voljo.

Iz rezultatov je razvidno, da je vpliv časa minimalno povezan z opravljanjem naših meritev. Kljub temu se pozna malenkost večji povprečni čas $T_{service}$ ob 14:00, 17:00, 22:00 in 05:00. Prav tako so ob tistih časih najbolj ekstremni največji časi storitve. Rekordni največji čas obdelave na strežniku je bil izmerjen ob 22:00, ki je znašal 3755.00 ms. V tem primeru je bilo povprečje malenkost povečano, znašalo je 318.68 ms. Rekord povprečja je bil izmerjen ob 05:00 in je znašal 548.89 ms. Takrat je recimo največji čas znašal 3590.00 ms. Rekord za najmanjši čas obdelave držita testa ob 19:00 in 21:00, takrat je najmanjši čas obdelave znašal 106 ms.

### 2.5.6 Peti test - neprestano pošiljanje

Pri vseh testih do sedaj smo strežnik obremenjevali v nekih določenih in omejenih intervalih. Pri petem testu smo želeli strežnik obremeniti v daljšem časovnem intervalu. Kot smo že omenili v razdelku 2.1.2, nam baza omogoča

maksimalno 10,000 zapisov (brez omejitve prostora). Pri testu lahko torej neprestano pošiljamo 10,000 datotek (oz. 9,990 - v mislih moramo imeti, da nekaj vrstic baze že zasedajo podatki o prijavi uporabniškega računa) ali pa jih pošljemo več, pri čemer bazo med samim oddajanjem sproti brišemo. Ker nas je zanimalo, ali se bo $T_{service}$ zvišal skozi čas ob že zasedeni bazi, smo se odločili za prvo možnost - pošiljanje brez brisanja. Glede na vse prejšnje rezultate testov, smo za čas oddajanja izbrali 3 sekunde, način oddajanja je bil Poisson, .png datoteke velikosti 1,1 MB pa so bile zgenerirane z isto bash skripto kot pri vseh prejšnjih testih. Glede na okvirne izračune (na podlagi časov prejšnjih testov) naj bi tak test trajal okoli 9 ur, zato smo se odločili, da test izvedemo v noči iz 26. na 27. april 2019, na lokaciji 2.

### Hipoteza

Predpostavljamo, da se bo $T_{service}$ skozi čas večal, saj se bo velikost baze ves čas povečevala, kar bo vplivalo na čas shranjevanja.

### Rezultati

Zjutraj smo ugotovili, da se je sredi noči sesul brskalnik Chrome, zato smo test ponovili še enkrat. Tudi v soboto, 27. aprila, se je pri ponovnem poskusu testiranja brskalnik sesul. Baza je takrat glede na podatke iz Herokujeve nadzorne plošče vsebovala 2902 vrstic, pri čemer je velikost baze znašala 3,9 GB. Po nekaj prebranih zadetkih na Googlovem iskanju smo ugotovili, da imajo brskalniki vklopljen maksimalni limit pomnilnika, ki ga lahko porabijo, ta pa naj bi znašal okoli 3,5 GB. Takoj nam je bilo jasno, da je to krivec za naš problem. Po večih poskusih smo ugotovili, da lahko pride do sesutja že pri 3000 datotekah. Po pregledu dnevniških datotek smo ugotovili, da so zadnje naložene datoteke imele praktično identični čas obdelave kot pri vseh prejšnjih testih.

Ker intervalnega pošiljanja nismo želeli razbiti v več intervalov z vmesnim presledkom par minut (da se rezultati obdelajo in jih mi lahko shranimo), smo se odločili, da bomo število datotek zmanjšali na 2500.

Tudi pri testu s 2500 datotekami ni šlo brez težav. Ob sprotnem spremljanju dnevniških datotek je sam prenos in obdelava potekala brez problema, do napake z oznako 413 pa je prišlo po tem, ko smo na strežnik poslali zahtevo za prikaz rezultatov testiranja. Za tokratno napako smo bili krivi sami, saj smo imeli limit za število parametrov v aplikacijski prošnji (ang. `request`) nastavljen na 1000. Problem smo odpravili v nedeljo, 28.04.2019 in najprej izvedli test za 500 datotek. Rezultati testa so vidni v tabeli 2.10.

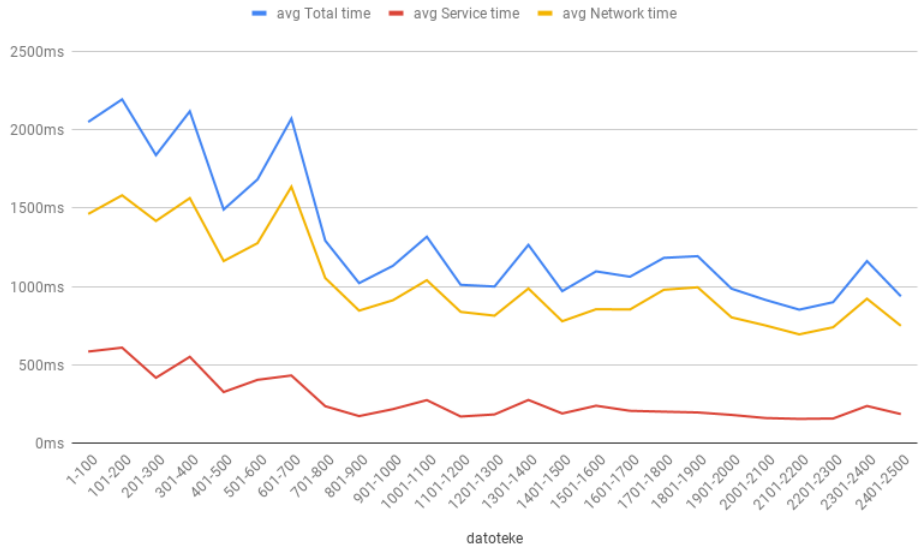
| | $T_{total}$ | $T_{service}$ | $T_{network}$ | RAM |
|---|---|---|---|---|
| avg | 1068.20 ms | 156.15 ms | 912.05 ms | 201.72 MB |
| max | 2509.00 ms | 706.00 ms | 1824.00 ms | 240.12 MB |
| min | 634.00 ms | 118.00 ms | 505.00 ms | 166.31 MB |

Tabela 2.10: Rezultati testa 5-1 - 500 datotek.

Potem smo ob 13:30 (prav tako v nedeljo, 28.04.2019) izvedli test z 2500 datotekami. Tabelarični rezultati testa 5-2 so vidni v tabeli 2.11, grafični pa na sliki 2.11. Graf smo sestavili tako, da smo ročno izračunali vse 3 čase za sklope po 100 datotek, od 0 do 2500.

| | $T_{total}$ | $T_{service}$ | $T_{network}$ | RAM |
|---|---|---|---|---|
| avg | 1310.64 ms | 280.84 ms | 1029.80 ms | 214.33 MB |
| max | 11907.00 ms | 7777.00 ms | 6933.00 ms | 273.57 MB |
| min | 631.00 ms | 106.00 ms | 522.00 ms | 158.49 MB |

Tabela 2.11: Rezultati testa 5-2 - 2500 datotek.

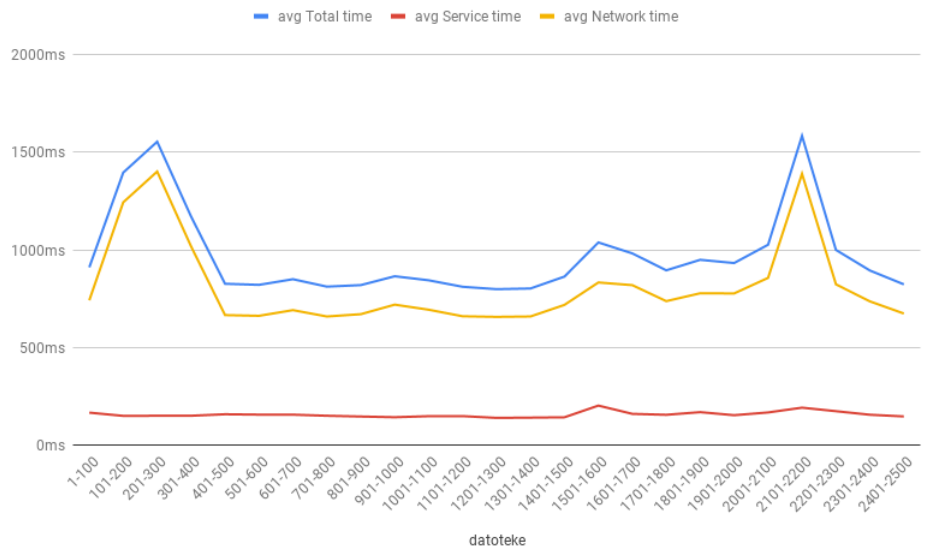



Slika 2.11: Rezultati testa 5-2 - 2500 datotek.

Izkaže se, da so bili vsi trije časi ob začetku izvajanja testa precej veliki. Ker razloga za porast časov nismo našli smo se odločili, da test ob 17:30 (prav tako v nedeljo, 28.04.2019) ponovno ponovimo. Tabelarični rezultati testa 5-3 so vidni v tabeli 2.12, grafični pa na sliki 2.12.

| | $T_{total}$ | $T_{service}$ | $T_{network}$ | RAM |
|---|---|---|---|---|
| avg | 971.10 ms | 159.01 ms | 812.09 ms | 207.72 MB |
| max | 2710.00 ms | 1154.00 ms | 2448.00 ms | 246.21 MB |
| min | 647.00 ms | 105.00 ms | 515.00 ms | 158.09 MB |

Tabela 2.12: Rezultati testa 5-3 - 2500 datotek.




Slika 2.12: Rezultati testa 5-3 - 2500 datotek.

**Ugotovitve**

Pri tretjem testu v razdelku 2.5.4 smo ugotovili, da povprečni $T_{service}$ za čas oddajanja 3.5 po Poissonu znaša 267.34 ms. Ko smo v testu 2.10 izvedli pošiljanje 500 datotek, je povprečni $T_{service}$ znašal 156.15 ms, kar je celo manj kot za samo 100 datotek. Razloga za to nismo našli. Izvedba testa 5-2 je znašala 2 uri in 26 minut, medtem ko izvedba testa 5-3 2 uri in 27 minut. Menimo, da smo v tem času strežnik dovolj obremenili, da bi lahko pokazal določene anomalije. Ker rezultatov testa 5-2 ne znamo interpretirati, bomo pokomentirali zgolj rezultate testa 5-3. Celoten povprečni čas $T_{service}$ je znašal 159.01 ms, kar je zgolj 2.86 ms več, kot pri 500 datotekah (kar je manj kot 10% razlike). Menimo, da je strežnik uspešno prestal test neprestanega pošiljanja, saj na grafu 2.12 jasno vidimo praktično konstanten povprečni čas obdelave.

### 2.5.7 Šesti test - bremenski test

Pri vseh dosedanjih testih smo strežnik obremenili iz zgolj enega odjemalca. Seveda nas je zanimalo, kako bi se obnašal odjemalec v primeru, da ga obremenimo iz večih odjemalcev hkrati (kot se to dogaja v realnem okolju). Odločili smo se, da bomo bremenski test izvedli tako, da strežnik obremenimo iz lokacije 2 in 3 hkrati. Da bi pridobili čim več podatkov za analizo, smo za test izbrali iste nastavitve kot pri petem testu v prejšnjem razdelku (2.5.6). To pomeni različnih 2500 datotek (velikost 1,1 MB, tipa .png) iz vsakega odjemalca, čas

oddajanja 3 sekunde in način oddajanja datotek po Poissonu. Test smo izvedli v četrtek, 02.05.2019 ob 21:42.

### Hipoteza

Predpostavljamo, da se bo v primerjavi s petim testom (razdelek 2.5.6) $T_{service}$ povečal, ker bo strežnik bolj obremenjen.

### Rezultati

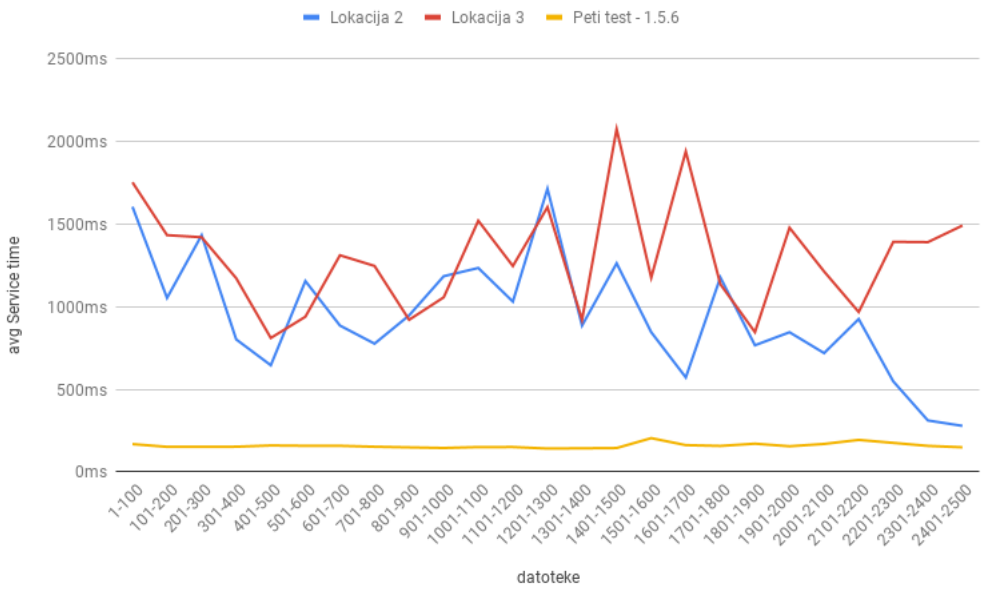
| | $T_{total}$ | $T_{service}$ | $T_{network}$ | RAM |
|---|---|---|---|---|
| avg | 2612.95 ms | 944.12 ms | 1668.84 ms | 256.41 MB |
| max | 16491.00 ms | 9764.00 ms | 13010.00 ms | 359.09 MB |
| min | 814.00 ms | 123.00 ms | 643.00 ms | 171.07 MB |

Tabela 2.13: Rezultati testa 6-1 - odjemalec na lokaciji 2.

| | $T_{total}$ | $T_{service}$ | $T_{network}$ | RAM |
|---|---|---|---|---|
| avg | 18883.27 ms | 1298.23 ms | 17585.04 ms | 259.23 MB |
| max | 214191.00 ms | 10524.00 ms | 212605.00 ms | 341.88 MB |
| min | 2704.00 ms | 136.00 ms | 2542.00 ms | 171.07 MB |

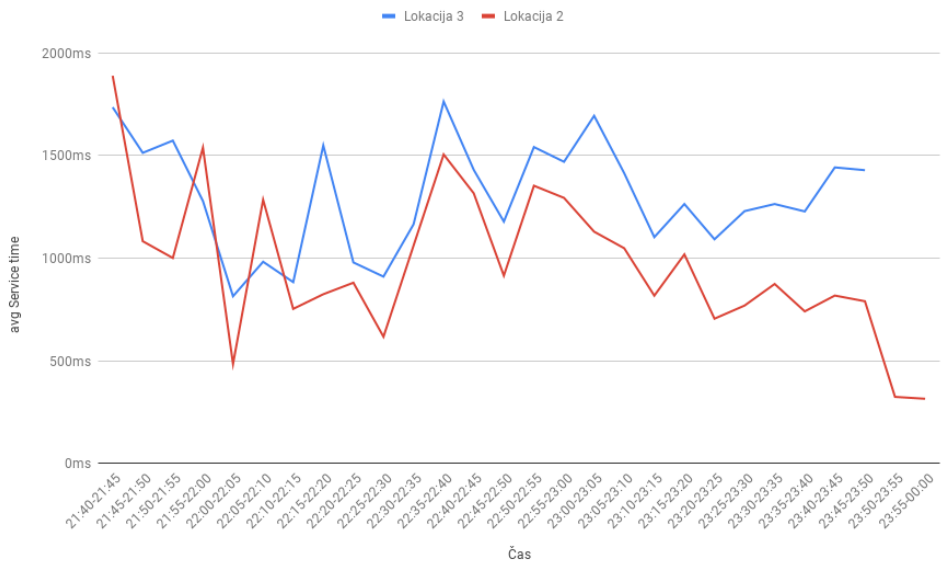
Tabela 2.14: Rezultati testa 6-2 - odjemalec na lokaciji 3.

Po pregledu rezultatov v tabelah 2.13 in 2.14 jasno vidimo, da je prišlo do ogromnega porasta $T_{service}$. Prišlo je tudi do nekaj večjega $T_{network}$, kar skupaj s $T_{service}$ vodi v ogromen porast $T_{total}$. Ta je kar enkrat večji v primerjavi z rezultati petega testa (razdelek 2.5.6). Grafični prikaz rezultatov je viden na sliki 2.13, narejen pa je na enak način kot slika 2.11 v petem testu. Na grafu vidimo povprečen čas obdelave datotek za oba odjemalca, za primerjavo pa smo dodali tudi rezultate testa 5-3, vidne v tabeli 2.12.

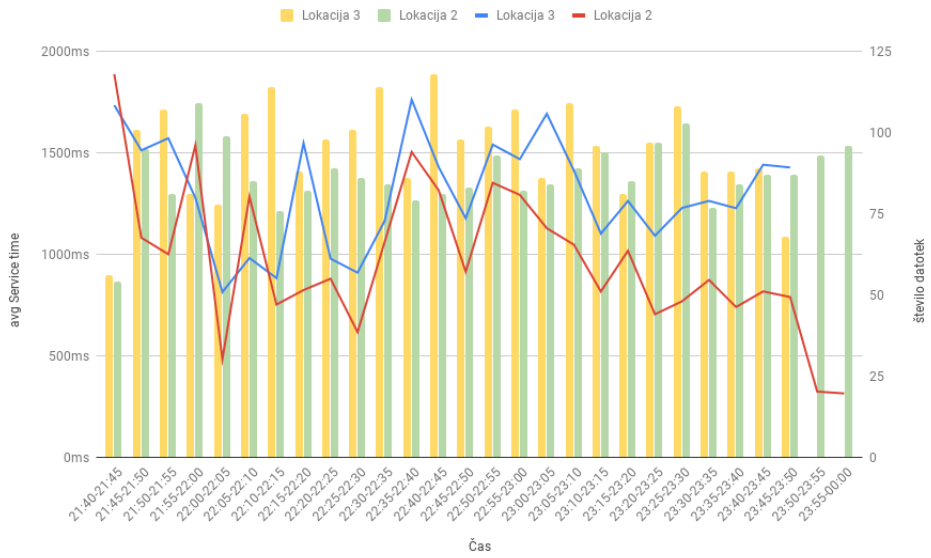



Slika 2.13: Rezultati testa 6 - prikaz $T_{service}$ za oba odjemalca hkrati in primerjava z rezultati testa 5-3 iz razdelka 2.5.6.

Po podrobnem pregledu rezultatov smo ugotovili, da sta se testa končala ob različnem času, kar smo zaradi Poissonove porazdelitve pravzaprav pričakovali. Test na lokaciji 2 je trajal 2 uri in 20 minut, test na lokaciji 3 pa 2 uri in 8 minut. To predstavlja 12 minutno razliko, kar je skoraj 9%. Zaradi tega iz zgornje slike 2.13 ne moremo točno ugotoviti, ali je povprečni $T_{service}$ v določenem času nihal enako za oba odjemalca. Rezultate smo v nadaljevanju obdelali tako, da le-ti predstavljajo povprečni $T_{service}$ v enakih časovnih intervalih. Prikazani so na sliki 2.14. Na sliki 2.15 prikažemo isti graf kot na sliki 2.14, le da smo tu dodali tudi število datotek, ki sta jih poslala oba odjemalca v istem časovnem intervalu.




Slika 2.14: Rezultati testa 6 - prikaz $T_{service}$ za oba odjemalca hkrati v istih časovnih intervalih.




Slika 2.15: Rezultati testa 6 - prikaz $T_{service}$ za oba odjemalca hkrati v istih časovnih intervalih s številom datotek v določenih časovnih intervalih.

Po pregledu slike 2.14 vidimo, da so bili časi $T_{service}$ za oba odjemalca sprva povsem različni. Po približno 30 minutah sta se časa umirila, vidi pa se enakomerno nihanje $T_{service}$ za oba odjemalca. Skozi čas vidimo, da so bile zahteve s strani odjemalca na lokaciji 2 obdelane v krajšem času kot pa tiste iz odjemalca na lokaciji 3. Na koncu jasno vidimo občuten padec $T_{service}$ na lokaciji 2 po tem, ko se je oddajanje na lokaciji 3 zaključilo. Konkretnega razloga (zakaj je na začetku $T_{service}$ večji, potem pa se postopoma umiri) nismo našli, slutimo pa, da Heroku na začetku ne dodeli vseh razpoložljivih procesnih virov strežniku dokler ta ni dovolj obremenjen (po Herokujevih pravilih). Ta ugotovitev bi razložila tudi anomalijo, ki je nastala pri prvem poizkusu petega testa (razdelek 2.5.6, natančneje Rezultati testa 5-2 - slika 2.11). Spremenljivo dodelitev razpoložljivih procesnih virov strežnika lahko utemeljimo tudi s sliko 2.15, kjer porast števila datotek več kot očitno ni bil edini faktor za porast $T_{service}$.

### Ugotovitve

Po pregledu rezultatov smo ugotovili, da lahko zastavljeno hipotezo potrdimo. Izkaže se, da se je povprečni $T_{service}$ za odjemalca na lokaciji 2 v primerjavi s petim testom (druga ponovitev, tabela 2.12) povečal iz 159.01 ms na 944.12 ms, kar je 494% povečanje. Sprememba minimalnega povprečnega časa znaša 18.00 ms, kar je 17% povečanje, sprememba maksimalnega povprečnega časa pa znaša 8610.00 ms, kar je 746% povečanje. Za lokacijo 3 ne moremo narediti direktne

primerjave, saj iz nje ni bil izveden peti test. Kljub temu lahko odjemalca iz lokacije 3 primerjamo z odjemalcem iz lokacije 2. Tu zaznamo še dodano povečanja povprečnega $T_{service}$ za kar 354.11 ms. Razlika v minimalnem in maksimalnem $T_{service}$ je v tem primeru manjša kot 10%.

Rezultati tega testa so nas precej presenetili, saj nismo pričakovali spet tako velikega porasta za $T_{service}$. Ugotavljamo, da je bil cilj bremenskega testa uspešno dosežen.

### 2.5.8 Sedmi test - stress test

Pri sedmem testu smo se odločili, da strežnik do limita obremenimo z vidika pomnilnika. Po podrobnem pregledu Herokujevih specifikacij smo opazili, da nam v osnovnem (brezplačnem) načinu ponuja 500 MB pomnilnika, ki ga lahko preobremenimo do dodatnih 500 MB (ki delujejo po principu swap pomnilnika). Heroku aplikacijo ugasne takoj, ko je presežen limit 1 GB (500 MB RAM + 500 MB swap). Aplikacija se takrat sploh ne odziva, nazaj pa se postavili šele po nekaj minutah ali takoj po tem, ko je ročno sprožen ponovni zagon. Da bi aplikacijo dodano obremenili, smo morali umetno generirati dodatno zasedenost pomnilnika. Po razmisleku smo ugotovili, da lahko izvedemo dva tipa stress testa. V prvem načinu (poimenujemo ga način A) lahko povečamo obremenitev RAM-a direktno na strani strežnika (deklariranje velike vrednosti v neko spremenljivko), v drugem (način B) pa posledično - preko baze (deklariranje velike vrednosti v neko spremenljivko, pri čemer je vrednost spremenljivke pridobljena iz baze). Ker obremenitev pomnilnika (zaradi zasnove aplikacije) ni možna konstantno skozi čas, smo obremenitev izvedli pri vsakem sprotnem zahtevku za obdelavo oz. shrambo naložene datoteke. Samo realizacijo obremenitve lahko postavimo pred ali za štoparico $t_2$ (opis štoparic je razviden v razdelku 2.4), kar lahko vpliva na prikaz časov. Prvega poimenujmo način BEFORE, drugega pa način AFTER. Če izvajanje obremenitve postavimo pred štoparico $t_2$, bo povečan čas za generiranje stresa na zasedenost pomnilnika vključen v $T_{network}$, če pa izvajanje obremenitve postavimo za štoparico $t_2$, pa bo le-ta vključen v $T_{service}$. Po tehtnem premisleku in dejstvu, da bo v tem primeru težko primerjavi $T_{service}$ s preteklimi testi, smo se odločili, da implementiramo dodatno štoparico za način BEFORE, poimenovano $t_{1.5}$. $T_{service}$ se tako še vedno izračuna kot $t_3$ - $t_2$, dodamo pa lahko nov čas - $T_{stress}$, ki ga izračunamo kot $t_2$ - $t_{1.5}$. Zaradi tega je posledično spremenjen tudi izračun $T_{upload}$, ki je sedaj izračunan kot $t_1$ - $t_{1.5}$, $T_{network}$ pa kot $T_{total}$ - $T_{service}$ - $T_{stress}$.

Da bi zadostili obema možnima primeroma obmremenitve pomnilnika smo se odločili, da bomo to izvedli z sliko, zapisano kot niz tipa `base64`. V načinu A lahko dovolj veliko sliko zgeneriramo zgolj v zanki, čigar obremenitev bo definitivno vplivala na $T_{service}$, kar je še dodaten razlog za implementacijo dodatne štoparice. V načinu B bo datoteka priklicana iz baze, vedno pa bo šlo za enako datoteko. Da bo strežnik dodatno sliko res držal v pomnilniku skozi celoten čas $T_{service}$, smo pred vračanjem rezultata s strežnika implementirali klic na prvi znak iz `base64` niza datoteke. S tem poskrbimo, da datoteka ostane v pomnilniku.

Po pregledu rezultatov vseh prejšnjih testov smo opazili, da strežnik nikoli ne zahteva več kot cca. 350 MB RAM-a za obdelavo posameznih zahtev. Odločili smo se, da stresni test pričnemo z obremenitvijo velikosti 150 MB. Izbrali smo 100 datotek velikost 1,1 MB, način oddajanja Poisson, čas oddajanja zahtev 3 sekunde (zaradi smiselnega primerjanja rezultatov s prejšnjimi testi).

V aplikacijo smo zaradi potrebe tega testa na odjemalčev prikaz dodali možnost izbire tipa testa (način A ali B in način BEFORE ali AFTER), vse opisane spremembe pa ustrezno implementirali tudi na zalednem delu aplikacije. Prav tako smo na odjemalca dodali možnost kreiranja poljubne datoteke tipa `base64` glede na vnešeno velikost.

### Hipoteza

Predpostavljamo, da bo stress test več kot očitno vplival na $T_{service}$ oz. $T_{total}$, zaradi pomanjkanja izkušenj na tem področju pa žal ne moremo sklepati za koliko.

### Rezultati

Teste smo iz lokacije 2 pričeli izvajati v petek, 10.05.2019. Izkaže se, da strežnik nikakor ni sposoben shranjevanja datotek v primeru, da ga dodatno obremenimo za 150 MB (ne glede na način A ali B). Skozi dan smo poskušali zmanjševati velikost dodatne zasedenosti pomnilnika in iskali razlog za problem sesutja. S testi smo nadaljevali v nedeljo, 12.05.2019 (prav tako iz lokacije 2). Po večih testih smo ugotovili, da je način AFTER za nas nezanimiv, saj nam ne pove ničesar konkretnega. Zaradi tega smo izvajali zgolj načina A in B v kombinaciji z načinom BEFORE. Po večih zmanjšanjih velikosti stresa, različnih kombinacijah časa oddajanja in števila datotek smo se odločili, da število datotek postavimo na fiksnih 100, spremenimo pa kombinacijo časa oddajanja in načina stresnega testa. Definirali smo dve skrajni velikosti stresa, to sta 50 MB in 98 MB. Do teh dveh vrednosti smo prišli s poskušanjem in razmislekom, da bi bili te testi smiselni. Čas oddajanja smo definirali kot 10 sekund, 8 sekund, 6 sekund, 4 sekunde in 3 sekunde.

Rezultati testov za velikost obremenitve 98 MB in način A so vidni v tabeli 2.15 in 2.16. V kolikor smo čas oddajanja povečali na 6 sekund je strežnik aplikacijo izklopil po tem, ko je bila poslana datoteka št. 48, pred izklopom pa je obdelal datoteko št. 44. Pri času oddajanja 4 sekunde je bila zadnja poslana datoteka št. 22, zadnja obdelana pa št. 10. Pri testu s časom oddajanja 3 sekunde je strežnik zavrnil že prvo datoteko. Teste za 6, 4 in 3 sekunde smo nato ponovili še 3-krat, sami rezultati pa niso vredni omembe, ker so bile razlike v primerjavi s prvotnimi rezultati minimalne.

| | $T_{total}$ | $T_{service}$ | $T_{stress}$ | $T_{network}$ | RAM |
|---|---|---|---|---|---|
| avg | 3032.36 ms | 635.33 ms | 1536.73 ms | 860.30 ms | 266.78 MB |
| max | 9198.00 ms | 6639.00 ms | 2924.00 ms | 3075.00 ms | 522.68 MB |
| min | 1770.00 ms | 138.00 ms | 973.00 ms | 526.00 ms | 220.66 MB |

Tabela 2.15: Rezultati testa 7-1 - način A, stress 98 MB, 100 datotek, čas oddajanja 10 sekund.

| | $T_{total}$ | $T_{service}$ | $T_{stress}$ | $T_{network}$ | RAM |
|---|---|---|---|---|---|
| avg | 2841.79 ms | 682.71 ms | 1096.01 ms | 1063.07 ms | 281.50 MB |
| max | 7209.00 ms | 5688.00 ms | 3752.00 ms | 3335.00 ms | 538.08 MB |
| min | 1347.00 ms | 120.00 ms | 627.00 ms | 527.00 ms | 220.75 MB |

Tabela 2.16: Rezultati testa 7-2 - način A, stress 98 MB, 100 datotek, čas oddajanja 8 sekund.

Pri testiranju načina B in stresu velikost 98 MB smo test zaključili že pri času oddajanja 10 sekund, saj je strežnik odneslo že pri datoteki št. 5.

Rezultati testov za velikost obremenitve 50 MB in način A so vidni v tabelah 2.17, 2.18, 2.19, 2.20 in 2.21.

| | $T_{total}$ | $T_{service}$ | $T_{stress}$ | $T_{network}$ | RAM |
|---|---|---|---|---|---|
| avg | 2088.87 ms | 370.97 ms | 770.30 ms | 947.60 ms | 206.40 MB |
| max | 5474.00 ms | 3106.00 ms | 1872.00 ms | 2209.00 ms | 240.46 MB |
| min | 1382.00 ms | 133.00 ms | 601.00 ms | 545.00 ms | 171.97 MB |

Tabela 2.17: Rezultati testa 7-3 - način A, stress 50 MB, 100 datotek, čas oddajanja 10 sekund.

| | $T_{total}$ | $T_{service}$ | $T_{stress}$ | $T_{network}$ | RAM |
|---|---|---|---|---|---|
| avg | 1930.94 ms | 328.71 ms | 553.17 ms | 1049.06 ms | 207.61 MB |
| max | 4165.00 ms | 1728.00 ms | 1194.00 ms | 2267.00 ms | 243.70 MB |
| min | 1182.00 ms | 138.00 ms | 309.00 ms | 568.00 ms | 186.61 MB |

Tabela 2.18: Rezultati testa 7-4 - način A, stress 50 MB, 100 datotek, čas oddajanja 8 sekund.

| | $T_{total}$ | $T_{service}$ | $T_{stress}$ | $T_{network}$ | RAM |
|---|---|---|---|---|---|
| avg | 3107.30 ms | 1264.30 ms | 705.39 ms | 1137.61 ms | 230.02 MB |
| max | 9564.00 ms | 7902.00 ms | 1386.00 ms | 5056.00 ms | 445.25 MB |
| min | 1560.00 ms | 224.00 ms | 596.00 ms | 544.00 ms | 181.17 MB |

Tabela 2.19: Rezultati testa 7-5 - način A, stress 50 MB, 100 datotek, čas oddajanja 6 sekund.

| | $T_{total}$ | $T_{service}$ | $T_{stress}$ | $T_{network}$ | RAM |
|---|---|---|---|---|---|
| avg | 1830.87 ms | 423.79 ms | 557.02 ms | 850.06 ms | 198.61 MB |
| max | 4618.00 ms | 1330.00 ms | 862.00 ms | 2975.00 ms | 305.30 MB |
| min | 1135.00 ms | 130.00 ms | 317.00 ms | 552.00 ms | 168.87 MB |

Tabela 2.20: Rezultati testa 7-6 - način A, stress 50 MB, 100 datotek, čas oddajanja 4 sekunde.

| | $T_{total}$ | $T_{service}$ | $T_{stress}$ | $T_{network}$ | RAM |
|---|---|---|---|---|---|
| avg | 2762.82 ms | 1103.79 ms | 496.43 ms | 1162.60 ms | 230.99 MB |
| max | 4330.00 ms | 2339.00 ms | 836.00 ms | 2886.00 ms | 371.39 MB |
| min | 1184.00 ms | 143.00 ms | 311.00 ms | 535.00 ms | 170.50 MB |

Tabela 2.21: Rezultati testa 7-7 - način A, stress 50 MB, 100 datotek, čas oddajanja 3 sekunde.

Pri testiranju načina B in stresu velikost 50 MB smo kot pri velikosti 98 MB naleteli na probleme z obdelavo (ugašanje strežnika). Za čas oddajanja 10 sekund je bila zadnja poslana datoteka št. 82, zadnja obdelana pa št. 72. Pri času oddajanja 8 sekund je bila zadnja poslana datoteka št. 84, zadnja obdelana pa št. 76. Pri času oddajanja 6 sekund je bila zadnja poslana datoteka št. 73, zadnja obdelana pa št. 61. Pri času oddajanja 4 sekunde je bila zadnja poslana datoteka št. 84, zadnja obdelana pa št. 55. Pri čas oddajanja 3 sekunde je bila zadnja poslana datoteka št. 91, zadnja obdelana pa št. 56. Kljub 3-kratni ponovitvi vseh teh testov ni prišlo do drastičnih sprememb pri samih rezultatih oz. času padca storitve.

**Ugotovitve**

Izkaže se, da aplikacija pod stresom ni sposobna obdelati vseh podanih datotek. Po podrobnem pregledu dnevniških zapisov in rezultatov smo ugotovili, da je bil zastavljeni stres na samo aplikacijo prevelik. Prezahtevne zahteve prihajajo hitreje, kot pa se pomnilnik aplikacije lahko sploh prazni. Nekatere zahteve tako aplikacija sprejme in prične z njihovo obdelavo ker ne more vedeti, da bo potem pognana v cikel izdelave ogromne vrednosti, ki jo more shraniti v spremenljivko. Pri načinu B je krivda v tem, da aplikacija poleg dodanega stresa na pomnilnik porabi še nekaj dodatnega pomnilnika, da lahko vodi povezavo na bazo in nazaj, kar sledi v še večjo obremenitev in še hitrejšo odpoved sistema. Glede na opisano zatorej lahko potegnemo sklep, da je za porazne rezultate večina krivde na strani prezahtevne implementacije aplikacije in ne na samemu strežniku.

Na podlagi rezultatov uspešnih testov lahko zastavljeno hipotezo potrdimo. Kljub temu smo presenečeni nad rezultati, ker tako slabih rezultatov nikakor nismo pričakovali. Izkaže se, da povečana zasedenost RAM-a poskrbi, da lahko strežnik naenkrat obdela (hrani v čakalni vrsti) manj zahtev, zaradi česar moramo povečati čas oddajanja, če želimo zagotoviti enak $T_{service}$ kot pri prejšnjih

testih.

### 2.5.9 Osmi test - test s spreminjajočim časom oddajanja

Namen testa s spreminjajočim časom oddajanja je v tem, da skozi čas spremljamo, kako se spreminja $T_{service}$ oz. $T_{total}$ v primerjavi s podanim oddajnim časom. Za potrebe testa smo morali poseči po spremembah v naši aplikaciji. Na odjemalcu smo dodali vnosno polje za začetno vrednost časa oddajanja. Definirali smo tudi vnosno polje za število datotek, pri katerem se spremeni čas oddajanja in vnosno polje za minimalen čas oddajanja, pri kateri se test še izvaja. Dodano je bilo tudi vnosno polje, ki ponazarja stopnico oz. faktor zmanjšanja. Za vsa vnosna polja je bila dodana ustrezna logika v zaledenem sistemu aplikacije. Z dodatno implementacijo lahko tako definiramo začeten čas oddajanja oddaja I, število datotek J (po katerih se čas oddajanja poviša), spremembo K (ki označuje faktor višanja časa oddajanja) in končnen čas oddajanja L (pri kateri se test preneha).

Test smo sprva zastavili tako, da smo začetni čas oddajanja postavili na 10 sekund, ga zmanjševali za 1 sekundo na vsakih 100 datotek in končali pri času 2 sekundi. Ker smo želeli ugotoviti čase pri spremembi stopnice oz. časa oddajanja smo način oddajanja nastavili na enakomerno oddajanje. Kot pri ostalih testih smo za pošiljanje izbrali datoteke velikosti 1,1 MB. Glede na rezultate prejšnjih testov in nekaj začenih poiskusih smo ugotovili, da je sprememba $T_{service}$ premalo očitna, zato smo test zastavili na manjše parametre.

Izbrali smo začetni čas oddajanja 1200 ms, minimalni čas oddajanja 400 ms in velikost stopnice 200 ms. Čas oddajanja smo zmanjšali za vsakih 400 datotek. Test smo izvajali dne 15.05.2019 okoli 17:00 iz lokacije 2.

#### Hipoteza

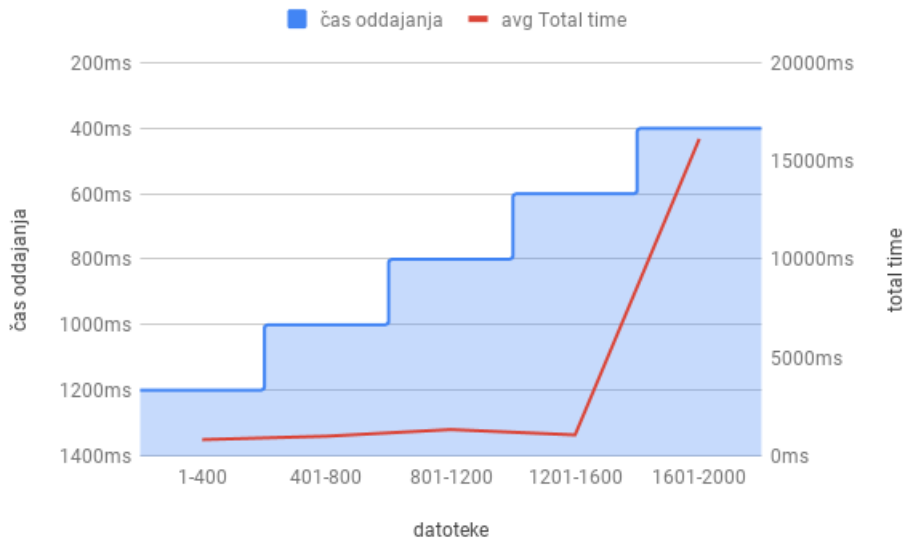
Pričakovali smo, da se bo z manjšanjem časa oddajanja povečeval $T_{service}$ in posledično tudi $T_{total}$.

#### Rezultati

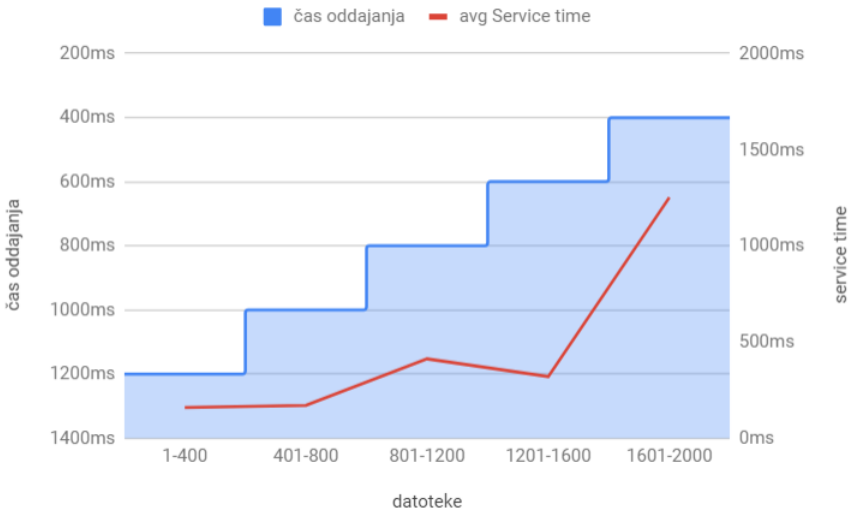
Rezultati testa so prikazani v tabeli 2.22. Grafični prikaz rezultatov je viden na slikah 2.16 in 2.17. Na grafih jasno vidimo potek višanja $T_{service}$ in $T_{total}$ glede na manjšanje časa oddajanja.

| | $T_{total}$ | $T_{service}$ | $T_{network}$ | RAM |
|---|---|---|---|---|
| avg | 4066.47 ms | 463.27 ms | 3603.20 ms | 221.18 MB |
| max | 33269.00 ms | 3009.00 ms | 31519.00 ms | 334.96 MB |
| min | 653.00 ms | 109.00 ms | 512.00 ms | 166.31 MB |

Tabela 2.22: Rezultati testa 8-1 - 2000 datotek s spreminjajočim časom oddajanja.




Slika 2.16: Rezultati testa 8-1 - prikaz $T_{total}$.




Slika 2.17: Rezultati testa 8-1 - prikaz $T_{service}$.

#### Ugotovitve

Kot smo pričakovali se z manjšanjem časa oddajanja povečuje $T_{service}$ in posledično tudi $T_{total}$. Iz grafov lahko vidimo, da se v začetku, ko je čas oddajanja še dovolj visok, $T_{service}$ in $T_{total}$ bistveno ne povečujeta. Z manjšim časom oddajanja, ko sistem postane bolj obremenjen, pa se $T_{service}$ in $T_{total}$ močno povečata.

### 2.5.10 Deveti test - test s spreminjajočim časom oddajanja po Poissonu

Deveti test predstavlja ponovitev prejšnjega (osmega testa) pri čemer je bil čas oddajanja definiran po Poissonovi porazdelitvi. Ostali parametri testa so bili enaki kot pri osmem testu, to pomeni: začetni čas oddajanja 1200 ms, minimalni čas oddajanja 400 ms, velikost stopnice 200 ms (tokrat po Poissonu) in sprememba v času oddajanja na vsakih 400 datotek.

Test smo izvajali dne 21.05.2019 okoli 12:00 iz lokacije 2.

#### Hipoteza

Pričakujemo, da bo povečanje $T_{service}$ (in posledično tudi $T_{total}$) še bolj izrazito kot pri prejšnjem testu.

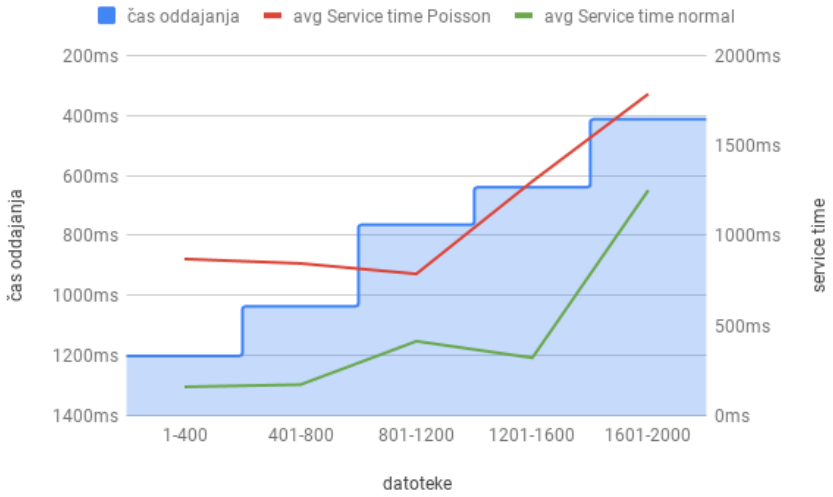
#### Rezultati

Rezultati testa so prikazani v tabeli 2.23. Grafični prikaz rezultatov je viden na slikah 2.18, 2.19 in 2.20. Zaradi nazornega prikaza rezultatov smo v grafih 2.18 in 2.19 za izbran interval povprečili čase za 400 datotek, na grafu 2.20 pa so prikazani posamični časi oddajanja za posamezno datoteko. Iz grafov jasno vidimo, da Poisson v povprečju izenači velikost stopnice, vendar zaradi velikega nihanja poslanih datotek poskrbi za narast $T_{service}$ in $T_{total}$ glede na rezultate prejšnjega testa.

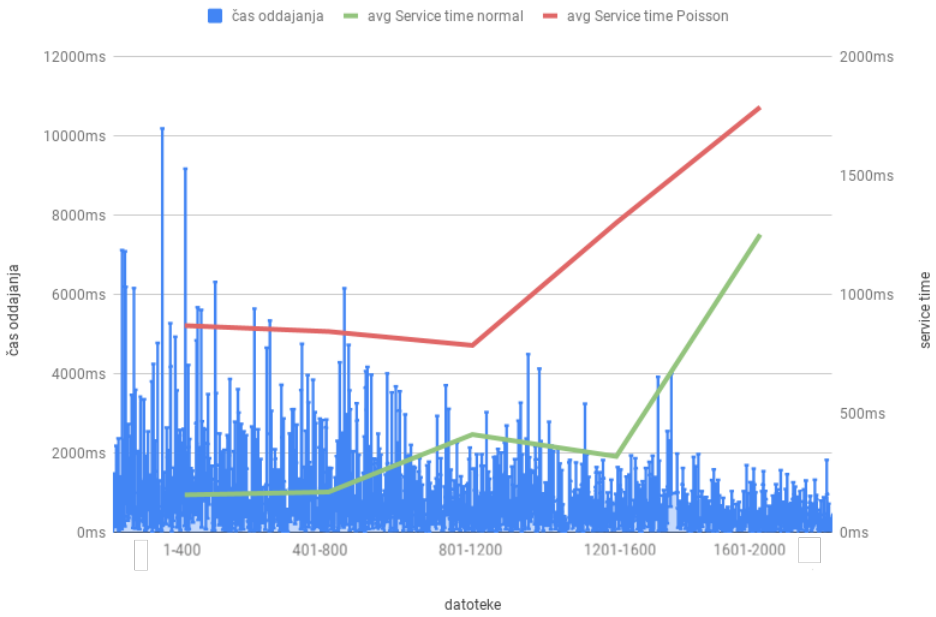
| | $T_{total}$ | $T_{service}$ | $T_{network}$ | RAM |
|---|---|---|---|---|
| avg | 9035.22 ms | 1117.84 ms | 7917.38 ms | 247.83 MB |
| max | 64779.00 ms | 6724.00 ms | 63342.00 ms | 354.71 MB |
| min | 662.00 ms | 116.00 ms | 500.00 ms | 166.27 MB |

Tabela 2.23: Rezultati testa 9-1 - 2000 datotek s spreminjajočim časom oddajanja po Poissonovi porazdelitvi.


Slika 2.18: Rezultati testa 9-1 - prikaz $T_{total}$ (rdeča krivulja) - 2000 datotek s spreminjajočim časom oddajanja po Poissonovi porazdelitvi (zelena krivulja predstavlja rezultate testa 8-1).




Slika 2.19: Rezultati testa 9-1 - prikaz $T_{service}$ (rdeča krivulja) - 2000 datotek s spreminjajočim časom oddajanja po Poissonovi porazdelitvi (zelena krivulja predstavlja rezultate testa 8-1).




Slika 2.20: Rezultati testa 9-1 - prikaz $T_{service}$ (rdeča krivulja) v primerjavi s posameznim časom oddajanja - 2000 datotek s spreminjajočim časom oddajanja po Poissonovi porazdelitvi (zelena krivulja predstavlja rezultate testa 8-1).

### Ugotovitve

Izkaže se, da test s Poissonovo porazdelitvijo v primerjavi z osmim testom pokaže narast $T_{service}$ že pri velikih časih oddajanja. Poleg tega opazimo narast pri $T_{total}$ , ki je za zadnjih 400 datotek v osmem testu znašal 16126,71 ms, medtem ko je po Poissonu ta čas narasel na 35104,23 ms. Hipotezo lahko zaradi tega zagotovo potrdimo.

## 2.6 Zaključek

Ugotavljamo, da je možnost brezplačnega gostovanja aplikacij na Herokuju povsem ustreza za preproste aplikacije, v našem primeru za pošiljanje, shranjevanje in prejemanje uporabnikovih datotek. Izkaže se, da je čas obdelave na strežniku zelo majhen, strežnik pa se je v večini testov obnašal precej konstantno. Največ težav je bilo na omrežju, kar je posledično povzročalo večje čase $T_{total}$. Strežnik je datoteke velikost 1,1 MB v večini primerov časovno obdelal skoraj v petini $T_{network}$, pri čemer je bila internetna povezava nadpovprečno hitra (100/100 Mbps). Ogromno povečanje $T_{service}$ je bilo zaznati šele v primeru, ko smo stre-

žnik bremensko obremenili. Do problemov je prišlo tudi v primeru stresnega testa, tam se je strežnik ob porastu pomnilnika na 500 MB precej upočasnil in dokaj hitro celo odpovedal. Testa s spreminjajočim časom oddajanja sta predstavljala testiranje skrajnih mej zmožnosti obdelave na strežniku.

Kljub vsemu v realni situaciji najverjetneje ne bi nikoli (glede na naše izkušnje) pošiljali 2500 slik zaporedoma iz večih odjemalcev hkrati, sploh pa ne obremenjevali strežnika za več kot 500 MB naenkrat. Tudi oddajanje na strežnik s spremenljivim časom oddajanja v realnosti ni nek realen scenarij, saj nas v realnosti največkrat omeji hitrost internetne povezave. Zaradi tega se nam Herokujevo brezplačno gostovanje zdi nadvse primerno za preproste aplikacije.

# Poglavje 3

# Analiza zmogljivosti oblačnih storitev

Gašper Bertoncelj, Janko Knez, Katja Kušar, Luka Mravinec

## 3.1 Opis storitve

Za analizo zmogljivosti oblačnih storitev smo si izbrali problem sortiranja števil, s katerim se danes pogosto srečamo na področju računalništva. Ker je problem enostaven za razumevanje in zanj obstaja veliko algoritmov, ki imajo različne časovne in prostorske zahtevnosti, je dobra metrika za hitrost prenosa podatkov in zmogljivost obdelave podatkov.

### 3.1.1 Node.js

Node.js je odprtokodna platforma [34], ki temelji na programskem jeziku JavaScript in je primarno namenjena pisanju strežniških aplikacij.

Pri implementaciji naše aplikacije se bomo obrnili na razširitev Node.js z knjižnico WebSocket [35][36][37], ki omogoča poenostavljeno povezavo klient-strežnik. Ponuja vrsto prestežitev za dogodke, katere lahko prožijo uporabniki ob prvi povezavi, pošiljanju datotek in prekinitvi povezave. S kombinacijo teh lahko implementiramo vse funkcionalnosti, ki jih mora naša aplikacija ponujati.

### 3.1.2 Python

Python je interpretiran, visokonivojski programski jezik [38], ki se pogosto uporablja za hiter razvoj prototipov aplikacij. V povezavi z Node.js je mogoče

ustvariti podproces programa Python, ki izvaja izbrano izvorno kodo. S tem lahko neodvisno od platforme Node.js kličemo jezike z različnimi zmogljivostmi.

Za potrebe aplikacije se bo uporabljal Python, v katerem se bodo implementirale metode za sortiranje. Te bodo klicane po prejetih zahtevah za sortiranje števil.

### 3.1.3 Scenarij uporabe

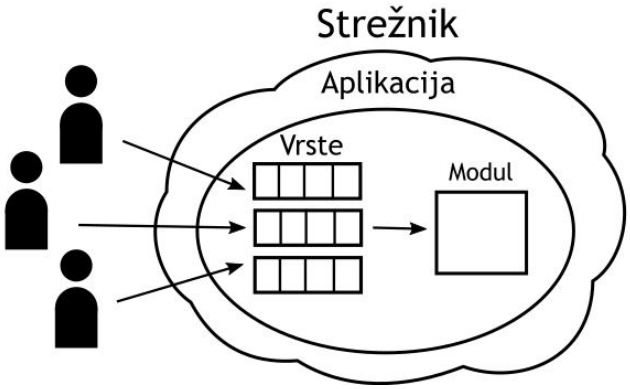
Aplikacijo, ki jo bomo napisali za potrebe naloge, bo tako omogočala prenašanje datotek med uporabnikom in strežnikom, hkrati pa sortirala števila, vsebovana v poslani datoteki. Uporaba aplikacije bo torej sledeča: uporabnik bo pri sebi generiral datoteke s števili, za lažje branje bo vsaka v svoji vrstici. Nato se bo preko konzole povezal z oddaljenim strežnikom, kateremu se pošiljajo zahteve, ki poleg izbranega jezika in metode za sortiranje vsebujejo tudi števila iz datoteke. Strežnik bo števila v prejeti zahtevi sortiral po velikosti in jih poslal nazaj k uporabniku. Pri pošiljanju zahtev bo uporabnik imel na voljo sinhrono oziroma čakajoče pošiljanje, kjer se nova zahteva ne pošlje preden predhodno poslana zahteva ni sprejeta nazaj, in asinhrono pošiljanje, kjer se zahteve strežniku pošiljajo s stalno frekvenco.

## 3.2 Realizacija storitve

Aplikacija za sortiranje števil je porazdeljena med uporabnikom in strežnikom: pri uporabniku zna pošiljati zahteve na strežnik in jih od tam tudi prejemati, aplikacija na strežniku pa zna sprejemati podatke od uporabnika, le-te urejati, in mu vračati rezultate urejanja.

Za realizacijo storitve smo se obrnili na platformo Node.js, kjer smo najprej napisali uporabniški del aplikacije. Vsa komunikacija z aplikacijo poteka preko konzole, preko katere se najprej generirajo datoteke - breme. Z generiranjem bremena vzpostavimo ponovljivost istega bremena ob različnih časih ali iz različnih lokacij. Ob generiranju se poda število ustvarjenih datotek in koliko števil je v vsaki datoteki.

Ob zaključenem generiranju se lahko prične pošiljanje zahtev strežniku. Na strežnik se lahko pošilja zahteve na dva načina: sinhrono in asinhrono. Pri asinhronem pošiljanju je potrebno poleg števila poslanih datotek podati aplikaciji tudi frekvenco oziroma čas zamika med zahtevami. S spreminjanjem le-tega bomo pri izvajanju samih eksperimentov dobili zmogljivost pri procesiranju velike kolicine zahtev. Zaradi narave kopičenja zahtev, če je frekvenca pošiljanja zahtev višja od frekvence procesiranja teh, smo na strežniku implementirali čakalno vrsto (podatkovna struktura vrsta) za vsakega uporabnika posebej. Pri tem je po poskusih na vsakega uporabnika najbolje imeti 3 podprocese Python, ki so zadolženi za sortiranje. V primeru ustvarjanja podprocesa za vsako novo zahtevo ob prejemu, bi strežniku hitro zmanjkalo delovnega pomnilnika. Po končanem sortiranju se uporabniku pošlje datoteka z odgovorom.




Slika 3.1: Shema pošiljanja zahtev v vrsto.

Kot je že zgoraj omenjeno, se bo za samo sortiranje uporabljal jezik Python, v katerem se bo za potrebe naloge izvajal algoritem vstavljanja [39]. Algoritem ali jezik je mogoče zamenjati z klicem drugega podprocesa in drugim vhodnim argumentom, ki določa izbrani algoritem.

Listing 3.1: Psevdokoda za algoritem sortiranja vstavljanje.

```
for i=1 to Array.length do:
    current = Array[i]
    j = i - 1

    while j >= 0 and current < Array[j] do:
        Array[i + 1] = Array[j]
           j = j - 1
    end while

    Array[j + 1] = current
end for
```

### 3.2.1 Opis bremena

Za testiranje zmogljivosti strežnika se mu bodo iz treh lokacij pošiljale zahteve. Pri tem moramo paziti na uravnoteženost posameznih testov, saj želimo hitreje priti do meje procesiranja velike količine manjših zahtev kot procesiranje majhne količine zahtev, ki so zahtevne. Za poskus preobremenitve strežnika bomo vzeli 100 in po potrebi tudi 1.000 števil, kjer bomo postopoma povečevali frekvenco pošiljanja zahtev in opazovali točko, kjer čas odgovora začne hitro naraščati, za število datotek pa 10, 100 in 10.000. Zgornjo in spodnjo mejo frekvenc bomo določili iz sinhronega testiranja in pridobljenih časov.

Pri zadnjih testih se bo frekvenca zahtev postopoma povečevala. Za boljši približek realnemu testu se bo vzela tudi variacija teh testov s Poissonovo porazdelitvijo. Za podporo generatorja števil iz Poissonove porazdelitvije se bo

uporabila knjižnica poisson-process [40], ki omogoča periodično klicanje funkcij, kjer so zamiki vzeti iz Poissonove porazdelitve.

### 3.2.2 Opis lokacij

Poskusi se bodo izvajali iz treh lokacij, predstavljene v tabeli 3.1.

| Lokacija | Ponudnik | Dohodna hitrost [mb/s] | Odhodna hitorst [mb/s] |
|---|---|---|---|
| Meniška vas | A1 | 10 | 0.768 |
| Srednja vas pri Polhovem Gradcu | A1 | 20 | 1 |
| Škofja Loka | Telekom | 100 | 50 |

Tabela 3.1: Ponudniki spletnih storitev in hitrosti, zakupljene iz testnih lokacij.

### 3.2.3 Amazon Web Services

Za gostovanje aplikacije smo si izbrali storitev Amazon Web Services [41], ki v svojem brezplačnem planu ponuja Intel Xeon E5-2676 v3, 1GB glavnega pomnilnika in 8GB prostora za shrambo. Preko nadzorne plošče za upravljanje računa smo pridobili IP naslov 35.177.60.125, ki se nahaja v Londonu, Veliki Britaniji. Na naslov 35.177.60.125 smo 27. 4. 2019 ob 21.23 poslali tracert in ping iz Škofje Loke.

Listing 3.2: tracert iz Škofje Loke.

```
1  Gateway.Home    Skofja Loka, Slovenia 14.427 ms
2  213.250.19.90   Ljubljana, Slovenia Telekom Slovenije d.d. 23.881 ms
3  95.176.242.66   Ljubljana, Slovenia Telekom Slovenije d.d. 27.739 ms
4  80.81.194.152   Frankfurt, Germany DE-CIX Management GmbH 43.927 ms
5  54.239.107.224     Seattle, USA Amazon Technologies Inc. 44.457 ms
6  54.239.107.171     Seattle, USA Amazon Technologies Inc. 33.819 ms
7  (no reply)
8  52.93.128.0          Seattle, USA Amazon Technologies Inc. 33.897 ms
9  52.95.61.158        Dublin, Ireland Amazon Technologies Inc. 36.464 ms
10 52.94.35.73          London, UK Amazon Technologies Inc. 38.485 ms
11 52.94.35.120        London, UK Amazon Technologies Inc. 34.754 ms
12 52.94.33.193        London, UK Amazon Technologies Inc. 35.019 ms
13 52.94.33.40          London, UK Amazon Technologies Inc. 34.876 ms
14 (no reply)
15 (no reply)
16 (no reply)
17 35.177.60.125      London, UK Amazon Data Services UK 26.020 ms
```

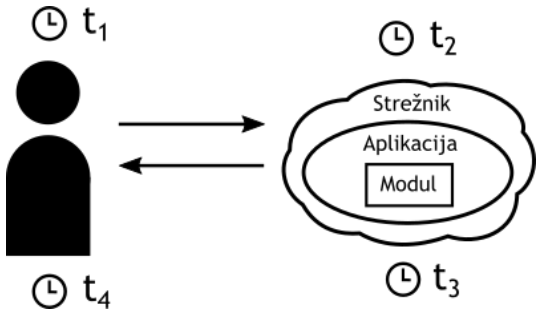
Listing 3.3: ping iz Škofje Loke.

```
PING 35.177.60.125 (35.177.60.125) 56(84) bytes of data.
64 bytes from 35.177.60.125: icmp_seq=1 ttl=45 time=32.9 ms
```

```
64 bytes from 35.177.60.125: icmp_seq=2 ttl=45 time=33.6 ms
64 bytes from 35.177.60.125: icmp_seq=3 ttl=45 time=44.4 ms
64 bytes from 35.177.60.125: icmp_seq=4 ttl=45 time=33.5 ms
64 bytes from 35.177.60.125: icmp_seq=5 ttl=45 time=32.3 ms
64 bytes from 35.177.60.125: icmp_seq=6 ttl=45 time=33.6 ms
64 bytes from 35.177.60.125: icmp_seq=7 ttl=45 time=33.3 ms
64 bytes from 35.177.60.125: icmp_seq=8 ttl=45 time=34.8 ms
64 bytes from 35.177.60.125: icmp_seq=9 ttl=45 time=32.8 ms
64 bytes from 35.177.60.125: icmp_seq=10 ttl=45 time=33.5 ms

--- 35.177.60.125 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9016ms
rtt min/avg/max/mdev = 32.386/34.516/44.426/3.368 ms
```

Po pridobljenem seznamu vmesnih skokov smo opazili skoke v Seattle, ki se nahaja na zahodni obali Združenih držav Amerike, ob večkratnem ponavljanju pa ni bilo odstopanja od prvega poskusa. Za ta pojav nimamo dobre razlage, ampak mislimo, da ima na to vpliv uporabljena storitev Amazon Web Services.

## 3.3 Metrike




Slika 3.2: Shema pošiljanja zahtevkov in izmerjeni časi.

### 3.3.1 Aplikacijske meritve

Zaradi porazdeljenosti aplikacije moramo čase meriti na različnih računalnikih, kjer pa pri samih izračunih ni potrebno skrbeti za sinhronizacijo ur. Zato na obeh straneh uvedemo štiri štoparice, ki hranijo uro, iz katerih bomo kasneje lahko izračunali potrebne čase. To so:

- $t_1$, ki hrani uro, ob kateri je bila iz uporabniške aplikacije poslana zahteva za urejanje.
- $t_2$, ki hrani uro, ko je bila zahteva sprejeta s strani aplikacije na oblačnem strežniku.
- $t_3$, ki hrani uro, ko je bil poslan odgovor s strežnika uporabniku.

- $t_4$, ki hrani uro, ko je uporabniška aplikacija sprejela odgovor strežnika.

Iz pridobljenih ur sedaj lahko izpeljemo:

- $T_{total} = t_4 - t_1$, ki nam pove čas, v katerem uporabnik prejme odgovor s strani aplikacije v obliki datoteke s sortiranimi števili.

- $T_{service} = t_3 - t_2$, ki nam pove čas, v katerem aplikacija zaključi sortiranje števil, začenši ob prejetju le teh.

- $T_{net} = T_{total} - T_{service}$, ki nam pove čas, potreben za pošiljanje zahteve in sprejem odgovora po omrežju.

Pri zadnjih testih je pomemben tudi $T_{delay}$, ki predstavlja zamik med zahtevami oziroma čas čakanja pri asinhronih testih preden se nova zahteva pošlje.

### 3.3.2 Strežniške meritve

Poleg zgoraj izračunanih časov smo med poskusi merili na strežniku tudi porabo procesorja in glavnega pomnilnika z ukazom `htop`, pri čemer smo morali strežnikovo aplikacijo pognati v ozadju.

Pridobljene meritve izrabe strežnika so bile pričakovane, kjer je bila izraba procesorja skoraj vedno 100%, glavni pomnilnik pa le 25% zaseden. Slednje je mogoče, ker se zahteve na strežniku kopičijo v vrsti in se streže le prvim trem vsakega uporabnika. Če bi vsako prejeto zahtevo začeli obdelovati ob prejemu, bi aplikacija hitro odpovedala, ko se zahteve začnejo kopičiti in mora za vsako novo ustvariti podproces za sortiranje. Veliko prostega delovnega pomnilnika nam zato dovoli ustvariti veliko količino zahtev, ki se lahko z majhnimi posledicami na zmogljivosti kopičijo v vrsti neobdelanih zahtev. Namen zadnjih poskusov pri nalogi je preobremeniti strežnik z veliko količino zahtev, ki porabijo celotni glavni pomnilnik ali prostor za hranjenje datotek.

## 3.4 Meritve in rezultati

Vsi rezultati meritev so prikazani v naslednjem odseku.

### 3.4.1 Sinhrono - pošiljanje 10 datotek s 100 števili

Pri testu se na strežnik pošlje 3-krat zahteva za sortiranje 10 datotek s 100 števili. Poskus je bil izveden iz vseh prej predstavljenih lokacij meritve, kjer so rezultati povprečje treh ponovitev. Za potrditev meritev smo isti test izvedli ob dveh različnih časih.

**Rezultati**

Prva izvedba poskusa je prikazana v sledečih tabelah.

| | $T_{total}$ [ms] | $T_{service}$ [ms] | $T_{net}$ [ms] |
|---|---|---|---|
| avg. | 83 | 28 | 55 |
| max. | 91 | 34 | 58 |
| min. | 81 | 27 | 53 |

Tabela 3.2: Meritev iz lokacije Srednja vas pri Polhovem Gradcu, 14. aprila 2019 ob 20:55.

| | $T_{total}$ [ms] | $T_{service}$ [ms] | $T_{net}$ [ms] |
|---|---|---|---|
| avg. | 69 | 27 | 41 |
| max. | 124 | 28 | 97 |
| min. | 59 | 27 | 31 |

Tabela 3.3: Meritev iz lokacije Škofja Loka, 14. aprila 2019 ob 22:29.

| | $T_{total}$ [ms] | $T_{service}$ [ms] | $T_{net}$ [ms] |
|---|---|---|---|
| avg. | 101 | 28 | 73 |
| max. | 162 | 33 | 129 |
| min. | 93 | 27 | 66 |

Tabela 3.4: Meritev iz lokacije Meniška vas, 15. aprila 2019 ob 00:00.

Za zaznavo možnih sistemskih obremenitev v času prvega testiranja smo celoten postopek ponovili ob drugem času. Druga izvedba poskusa je prikazana v sledečih tabelah.

| | $T_{total}$ [ms] | $T_{service}$ [ms] | $T_{net}$ [ms] |
|---|---|---|---|
| avg. | 93 | 27 | 66 |
| max. | 108 | 28 | 81 |
| min. | 81 | 27 | 54 |

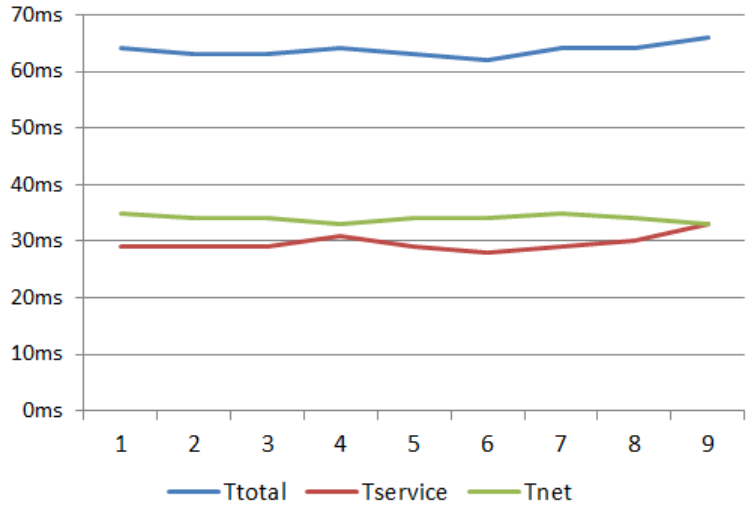
Tabela 3.5: Meritev iz lokacije Srednja vas pri Polhovem Gradcu, 21. aprila 2019 ob 21:21.

| | $T_{total}$ [ms] | $T_{service}$ [ms] | $T_{net}$ [ms] |
|---|---|---|---|
| avg. | 65 | 28 | 37 |
| max. | 85 | 33 | 57 |
| min. | 64 | 27 | 36 |

Tabela 3.6: Meritev iz lokacije Škofja Loka, 22. aprila 2019 ob 23:06.

| | $T_{total}$ [ms] | $T_{service}$ [ms] | $T_{net}$ [ms] |
|---|---|---|---|
| avg. | 116 | 27 | 89 |
| max. | 173 | 28 | 145 |
| min. | 90 | 27 | 64 |

Tabela 3.7: Meritev iz lokacije Meniška vas, 22. aprila 2019 ob 00:00.




Slika 3.3: Izmerjeni časi $T_{total}$, $T_{service}$ in $T_{net}$.

**Ugotovitve**
Za sinhrono pošiljanje so bili konstantni časi pričakovani, saj na strežniku ni možnosti kopičenja zahtev v vrsti. Čas $T_{service}$ je v povprečju enak 28Sms. Pridobljeni časi so torej idealni, kjer pa še ne vemo, kako se bo strežnik odzval na daljše izvajanje testa, kar bomo poskusili z naslednjim testom.

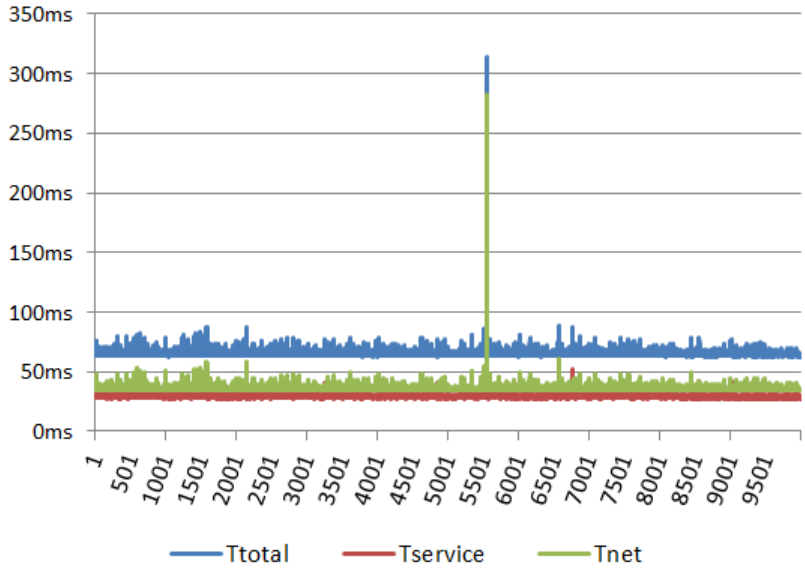
### 3.4.2 Sinhrono - pošiljanje 10.000 datotek s 100 števili

Pri testu se na strežnik pošlje zahteva za sortiranje 10.000 datotek s 100 števili. Glede na rezultate prejšnjega testa pričakujemo zelo podoben graf brez prevelikih odmikov od povprečja $T_{service}$ 30ms. Poskus je bil izveden iz Škofje Loke.

**Rezultati**

| | $T_{total}$ [ms] | $T_{service}$ [ms] | $T_{net}$ [ms] |
|---|---|---|---|
| avg. | 65 | 29 | 36 |
| max. | 314 | 52 | 282 |
| min. | 62 | 27 | 34 |

Tabela 3.8: Meritev iz lokacije Škofja Loka, 15. maja 2019 ob 20:00.




Slika 3.4: Izmerjeni časi $T_{total}$, $T_{service}$ in $T_{net}$.

**Ugotovitve**
Glede na način pošiljanja so rezultati pričakovani, torej zelo podobni prejšnjemu testu. Na podlagi tega lahko izpeljemo, da je $T_{service}$ povprečno 30ms tudi pri daljših sinhronih testih, kjer se v datotekah nahaja 100 števil in je izbran algoritem vstavljanja. Za asinhrone teste bomo torej zanesljivo vzeli 30ms kot čas najdaljše obdelave zahtev, kjer se le-te ne bodo kopičile v vrsti.

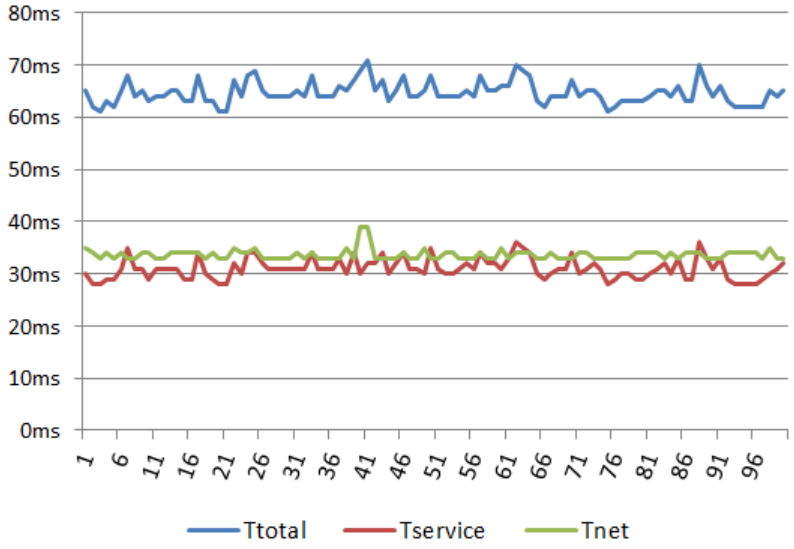
### 3.4.3 Asinhrono - pošiljanje 100 datotek s 100 števili s 30ms zamikom

Pri testu se na strežnik pošlje zahteva za sortiranje 100 datotek s 100 števili asinhrono, kjer se za zamik med zahtevami vzame ocena 30ms kot najnižja možna, preden $T_{service}$ začne rasti. Z drugimi besedami, na strežnik se bo neodvisno od prihoda odgovora na zahtevo, poslalo novo zahtevo vsakih pretečenih 30ms. Glede na rezultate prejšnjega testa pričakujemo zelo podoben graf brez prevelikih odmikov od povprečja. Poskus je bil izveden iz Škofje Loke.

**Rezultati**

| | $T_{total}$ [ms] | $T_{service}$ [ms] | $T_{net}$ [ms] |
|---|---|---|---|
| avg. | 64 | 31 | 33 |
| max. | 71 | 36 | 39 |
| min. | 61 | 28 | 33 |

Tabela 3.9: Meritev iz lokacije Škofja Loka, 16. maja 2019 ob 21:00.




Slika 3.5: Izmerjeni časi $T_{total}$, $T_{service}$ in $T_{net}$.

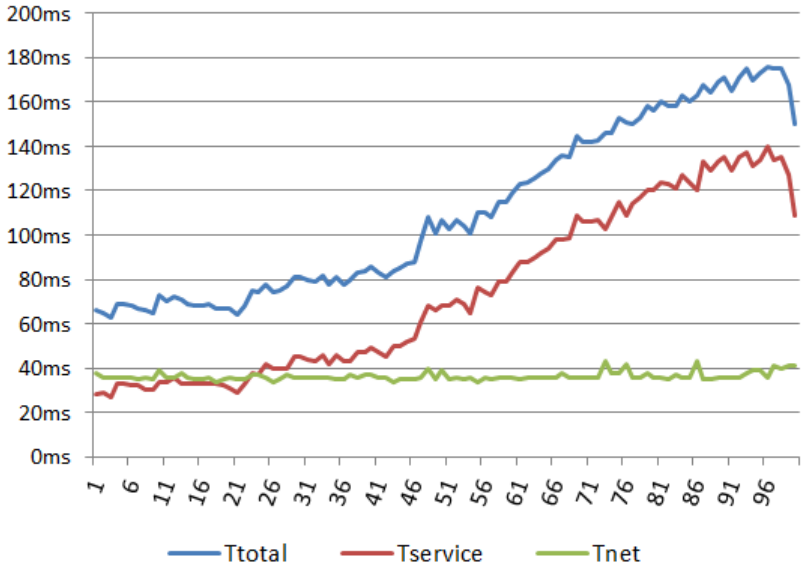
**Ugotovitve**
Tudi tukaj so rezultati pričakovani, čeprav smo sedaj pošiljali s stalno frekvenco, ki je na meji preden začne $T_{service}$ rasti. $T_{service}$ je torej enak kot pri sinhronih testih, kjer bomo z naslednjim testom poskusili postopoma zvišati $T_{service}$.

### 3.4.4 Asinhrono - pošiljanje 100 datotek s 100 števili z 28ms zamikom

Pri testu se na strežnik pošlje zahteva za sortiranje 100 datotek s 100 števili asinhrono, kjer se za zamik med zahtevami vzame ocena 28ms. Sedaj se glede na prejšnje rezultate pričakuje postopna rast $T_{service}$ in posledično tudi $T_{total}$. Poskus je bil izveden iz Škofje Loke.
**Rezultati**

| | $T_{total}$ [ms] | $T_{service}$ [ms] | $T_{net}$ [ms] |
|---|---|---|---|
| avg. | 110 | 74 | 36 |
| max. | 176 | 140 | 43 |
| min. | 63 | 27 | 34 |

Tabela 3.10: Meritev iz lokacije Škofja Loka, 17. maja 2019 ob 19:00.




Slika 3.6: Izmerjeni časi $T_{total}$, $T_{service}$ in $T_{net}$.

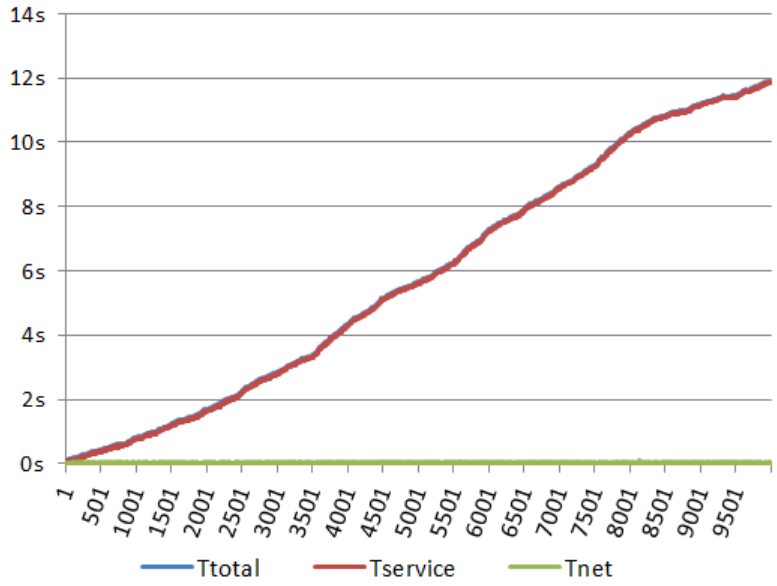
**Ugotovitve**
Po pričakovanjih začneta $T_{service}$ in $T_{total}$ postopoma rasti, čeprav iz grafov izgleda linearno. Za bolj pregledne čase pri asinhronem pošiljanju bomo sedaj poslali večje število datotek. Ob zadnji poslani zahtevi je opazen tudi padec časov $T_{service}$, saj za zadnji dve zahtevi ne potrebujemo vseh treh podprocesov Python in se lahko resursi procesorja razporedijo na manj zahtev.

### 3.4.5 Asinhrono - pošiljanje 10.000 datotek s 100 števili z 28ms zamikom

Pri testu se na strežnik pošlje zahteva za sortiranje 10.000 datotek s 100 števili asinhrono, kjer se za zamik med zahtevami vzame 28ms. Sedaj se glede na prejšnje rezultate pričakuje postopna rast $T_{service}$ in posledično tudi $T_{total}$, kjer ocenjujejo linearno rast, ki bo dosegla višje odzivne čase kot pri prejšnjem testu. Poskus je bil izveden iz Škofje Loke.
**Rezultati**

| | $T_{total}$ [ms] | $T_{service}$ [ms] | $T_{net}$ [ms] |
|---|---|---|---|
| avg. | 5.835 | 5.800 | 34 |
| max. | 11.910 | 11.877 | 92 |
| min. | 64 | 29 | 35 |

Tabela 3.11: Meritev iz lokacije Škofja Loka, 17. maja 2019 ob 19:00.




Slika 3.7: Izmerjeni časi $T_{total}$, $T_{service}$ in $T_{net}$.

**Ugotovitve**
Podobno kot pri prejšnjem asinhronem testu začneta $T_{service}$ in $T_{total}$ postopoma rasti. Kar pa je najbolj izrazito pri pridobljenem grafu je pa oblika krivulje, ki nastane. Časa $T_{service}$ in $T_{total}$ rasteta linearno. Po bolj podrobnem pregledu izvorne kode strežniške aplikacije lahko ta pojav potrdimo z razlago, da bo za n-to datoteko potreben dodaten $T_{addition}$ čas, ki se izračuna kot:
$T_{addition}$ = (n - 1) * ($T_{avg}$ - $T_{delay}$)

Za $T_{avg}$ se vzame glede na rezultate sinhronih testiranj 30ms, saj je to najpočasnejši povprečni odziv na zahtevo sortiranja 100 števil. $T_{delay}$ je lahko za veljavnost formule pozitivna vrednost, manjš od $T_{avg}$. To še dodatno potrdi dodatni pogled na graf, kjer se zamik med zahtevami razlikuje za 1-2ms od sinhronega povprečja, graf je pa temu primerno naklonjen.

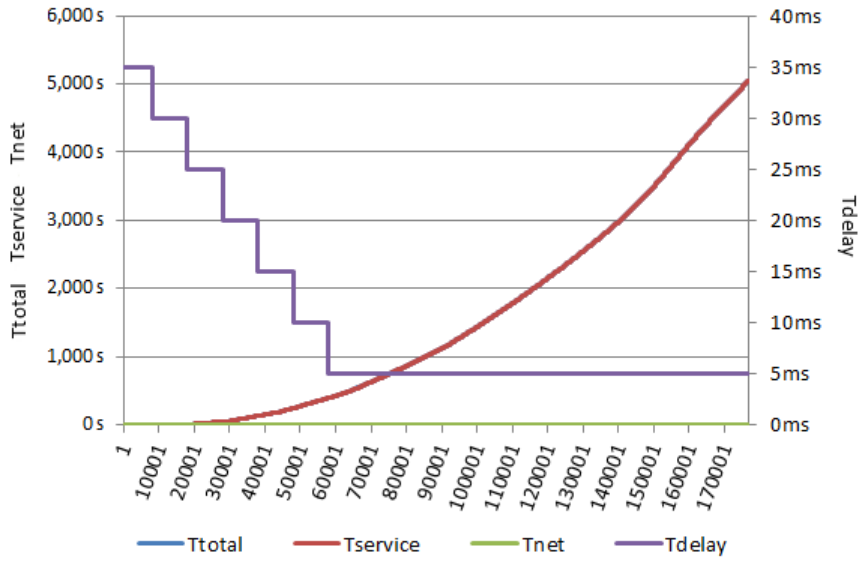
### 3.4.6 Asinhrono - postopno povečevanje frekvence pošiljanja vsakih 5 minut

Pri testu se na strežnik pošiljajo zahteve asinhrono. Začetni zamik med zahtevami je 35ms, ki se vsakih 5 minut zmanjša za 5ms. Test se bo izvajal dokler strežnik ni sposoben več sprejemati datotek. Zaradi same velikosti zahteve se pričakuje precej počasna poraba resursov sistema, ki v realnem primeru sama po sebi nima praktične uporabe, saj ne pričakujemo, da bo uporabnik neprestano pošiljal več 100.000 zahtev. Zaradi pričakovanja padca aplikacije bomo med izvajanjem testa merili količino prostega pomnilnika vsakih 5 minut. Poskus je bil izveden iz Škofje Loke.

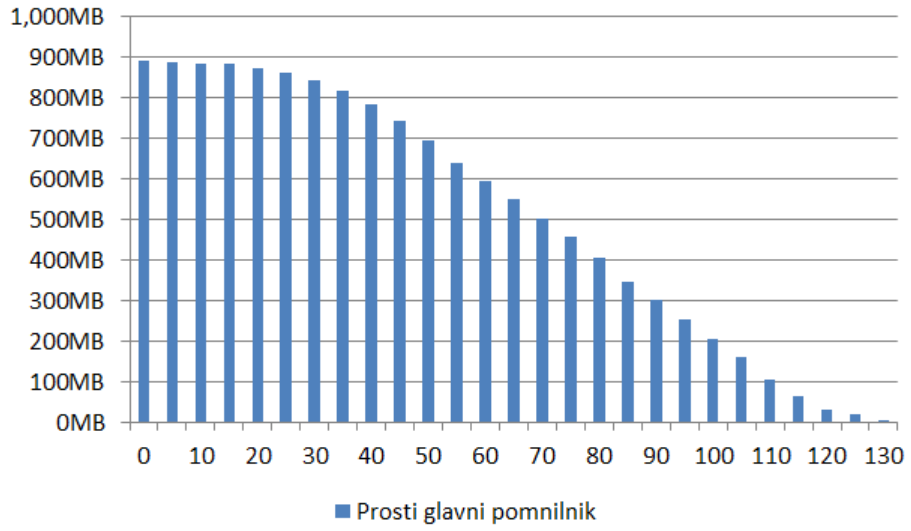
**Rezultati**

| | $T_{total}$ [s] | $T_{service}$ [s] | $T_{net}$ [s] |
|---|---|---|---|
| avg. | 1.563.128 | 1.563.089 | 39 |
| max. | 5.053.863 | 5.053.811 | 162 |
| min. | 61 | 26 | 19 |

Tabela 3.12: Meritev iz lokacije Škofja Loka, 22. maja 2019 ob 19:22.




Slika 3.8: Izmerjeni časi $T_{total}$, $T_{service}$ in $T_{net}$.




Slika 3.9: Meritve prostega glavnega pomnilnika.

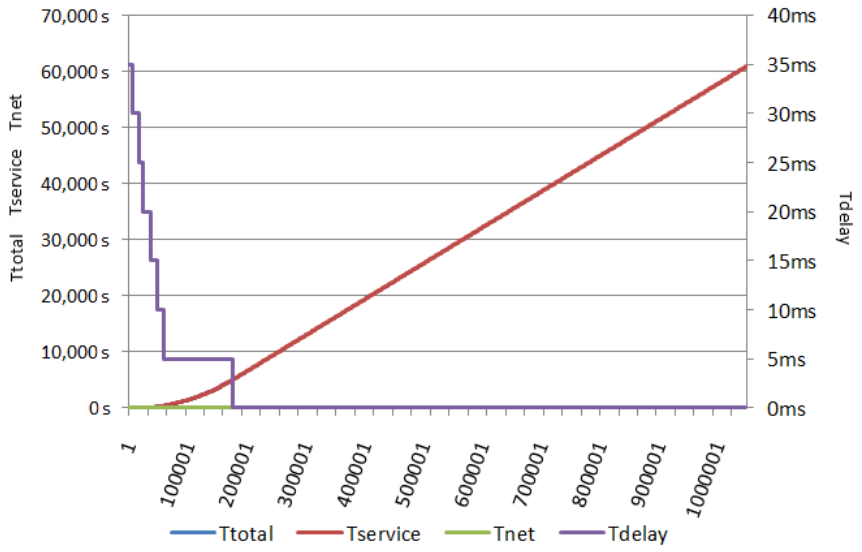
**Ugotovitve**
Test se je izvajal 2 uri in 11 minut, po katerem se je povezava z aplikacijo prekinila. Strežniku je od začetka prostega 891MB glavnega pomnilnika na koncu ostalo 0MB, zato aplikacija ni bila sposobna sprejemati nove zahteve ali ustvarjati podprocese Python, uporabniki pa izgubili povezavo s strežnikom. Graf prostega glavnega pomnilnika je pričakovane oblike, ki ima največji naklon ob prehodu 5ms $T_{delay}$. Na strežnik je bilo poslanih 1.051.609 zahtev, od katerih jih prišlo nazaj 176.955, torej je v vrsti čakalo 874.654 datotek s števili. Zahteve ob prekinitvi delovanja aplikacije skupaj porabljale le 419MB glavnega pomnilnika, ostalo porabo pripišemo podprocesom Python in ostalim uporabljenim spremenljivkam in neučinkovitosti jezika JavaScript za pomnjenje podatkovnih struktur. Ob začetku nismo pričakovali tako dolgega testiranja, ki je mogoče le zaradi izredno majhnih zahtev. Iz grafa je tudi razvidno, da je zaradi predhodnih neobdelanih zahtev bilo za zadnjo še uspešno odgovorjeno zahtevo potrebno počakati malo več kot 5000s od trenutka, ko je bila ta poslana.

### 3.4.7 Asinhrono - postopno povečevanje frekvence pošiljanja vsakih 5 minut s prekinitvijo pošiljanja

Kot nadaljevanje prejšnjega testa se bodo zahteve pošiljale asinhrono, kjer nas zanima kakšen graf prihoda odgovorov dobimo, če uporabnik preneha pošiljati nove zahteve tik preden aplikacija na strežniku odpove. Začetni zamik med zahtevami je 35ms, ki se vsakih 5 minut zmanjša za 5ms. Nove zahteve se bodo prenehale pošiljati, ko je na strežniku manj kot 25MB prostega glavnega pomnilnika. Graf, ki ga pricakujemo le podaljša krivuljo za $T_{service}$ prejšnjega

testa, ki bo naraščala hitreje kot linearno. Poskus je bil izveden iz Škofje Loke.

**Rezultati**




Slika 3.10: Izmerjeni časi $T_{total}$, $T_{service}$ in $T_{net}$.

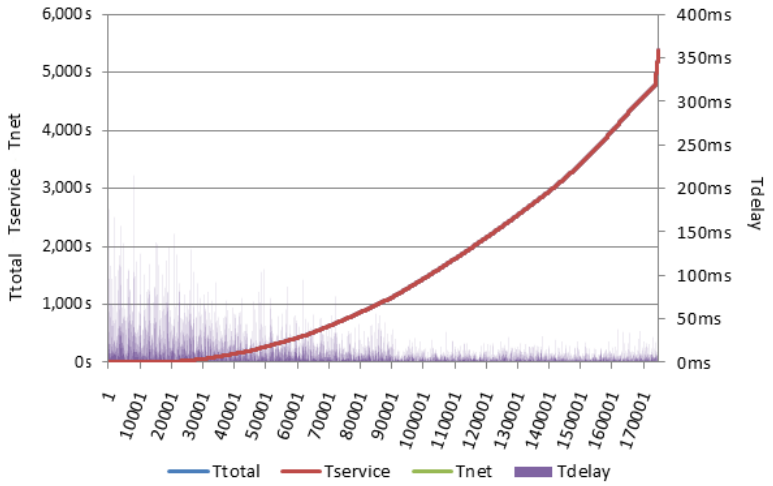
**Ugotovitve**

Test se je izvajal 21 uri in 33 minut, po katerem so od strežnika sprejeti vsi odgovori zahtev. Strežniku je od začetka prostega 891MB glavnega pomnilnika na koncu ostalo 793MB. Na strežnik je bilo v 2 urah in 5 minutah poslanih 1.043.000 zahtev, kjer je bilo obdelanih 170.563 zahtev preden je strežnik poslal uporabniku zahtevo za prenehanje pošiljanja novih zahtev. Od tega trenutka naprej je uporabnik samo sprejemal rezultate sortiranj. Linearnost krivulje $T_{service}$ pred izvedbo ni bila pričakovana, ampak jo lahko razložimo, da je trenutek, ko $T_{delay}$ postane 5ms v bližini dogodka, ko strežnik zahteva prenehanje pošiljanja novih zahtev.

### 3.4.8 Asinhrono - postopno povečevanje frekvence pošiljanja vsakih 5 minut s Poissonovo porazdelitvijo

Pri testu se na strežnik pošiljajo zahteve asinhrono. Začetni zamik med zahtevami je 35ms, ki se vsakih 5 minut zmanjša za 5ms. Dejanski zamik bo izračunan po Poissonovi porazdelitvi. Test se bo izvajal dokler strežnik ni možen več sprejemati datotek. Za generiranje zamikov se je uporabila knjižnica poisson-process [40], ki omogoča klicanje metod ob intervalih, vzetih kot števila iz Poissonove porazdelitve.

**Rezultati**




Slika 3.11: Izmerjeni časi $T_{total}$, $T_{service}$ in $T_{net}$.

**Ugotovitve**
Test se je izvajal 2 uri in 2 minut, po tem času se je povezava z aplikacijo prekinila zaradi pomanjkanja pomnilnika. Strežniku je od začetka prostega 891MB glavnega pomnilnika na koncu ostalo 0MB, zato aplikacija ni bila sposobna sprejemati nove zahteve ali ustvarjati podprocese Python, uporabniki pa izgubili povezavo s strežnikom. Na strežnik je bilo poslanih 1.080.290 zahtev, od katerih je prišlo nazaj 174.733, torej je v vrsti čakalo 905.557 datotek s števili. Izkaže se, da frekvence vzete iz Poissonove porazdelitve nimajo vpliva na odzivnost strežnika oziroma čas delovanja strežnika.

## 3.5 Zaključek

Namen naloge je bil analizirati zmogljivost oblačnih storitev s pomočjo problema sortiranja števil. Z rezultati, ki smo jih tekom testov pridobili, lahko svetujemo uporabo brezplačnih storitev za manjše potrebe, kjer količina zahtev nima tako velikega vpliva kot sama frekvenca pošiljanja oziroma pričakovanja odziva s strani uporabnika. Glavna izboljšava, ki jo lahko uporabnik naredi, je nakup več jeder na strežniku, saj obdelava zahtev večih uporabnikov naenkrat hitro upočasni odzivne čase. Za ugotavljanje hitrosti procesiranja zahtev je najprimernejši sinhroni test, kjer se hitro pridobi $T_{service}$, ki bo pri večjih zahtevah prispeval več k $T_{total}$ kot $T_{net}$. Za sinhrone in asinhrone zahteve, kjer je zamik med zahtevami $T_{delay}$ vsaj 30ms in se v datotekah nahaja 100 števil, se pričakuje $T_{service}$ 30ms, $T_{net}$ v rangu 20ms do 50ms, odvisno od povezave, in $T_{total}$ v rangu 50ms do 80ms. Brezplačni plan pri storitvi Amazon Web Servi-

ces se nam torej zdi primerna predvsem za posamezne uporabnik in za skupine uporabnikov, kjer rezultati niso takoj potrebni.

# Literatura


[1] "Amazon elastic compute cloud." https://aws.amazon.com/ec2/, 2019. [Online; dostopano 8-April-2019].

[2] "Aws free tier." https://aws.amazon.com/free/, 2019. [Online; dostopano 8-April-2019].

[3] "Amazon ec2 t2 instances." https://aws.amazon.com/ec2/instance-types/t2/, 2019. [Online; dostopano 8-April-2019].

[4] "Compression artifact." https://en.wikipedia.org/wiki/Compression_artifact/, 2019. [Online; dostopano 27-April-2019].

[5] "Amazon web services." https://aws.amazon.com/, 2019. [Online; dostopano 8-April-2019].

[6] "Launch a linux virtual machine." https://aws.amazon.com/getting-started/tutorials/launch-a-virtual-machine/?trk=gs_card, 2019. [Online; dostopano 8-April-2019].

[7] "Aws management console." https://aws.amazon.com/console/, 2019. [Online; dostopano 8-April-2019].

[8] "Aws management console." https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html, 2019. [Online; dostopano 8-April-2019].

[9] "Amazon elastic block store (amazon ebs)." https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AmazonEBS.html, 2019. [Online; dostopano 20-April-2019].

[10] "Lastnik ip naslova." https://db-ip.com/all/52.95.219, 2019. [Online; dostopano 1-Maj-2019].

[11] "Aws pricing calculator." https://calculator.aws, 2019. [Online; dostopano 4-Maj-2019].

[12] "Platform as a service." https://en.wikipedia.org/wiki/Platform_as_a_service, Marec 2019.

[13] "Cloud application platform | heroku." https://www.heroku.com/, Marec 2019.

[14] "Heroku pricing." https://www.heroku.com/pricing, Marec 2019.

[15] "Postgresql: The world's most advanced open source database." https://www.postgresql.org/, Marec 2019.

[16] "Node.js." https://nodejs.org/en/about/, Marec 2019.

[17] "Benchmarking cloud storage systems." https://brage.bibsys.no/xmlui/bitstream/handle/11250/263050/750269_FULLTEXT01.pdf?sequenc Julij 2014.

[18] "Base64 - wikipedia." https://en.wikipedia.org/wiki/Base64, Maj 2019.

[19] "expansion - base64: What is the worst possible increase in space usage? - stack overflow." https://stackoverflow.com/questions/4715415/base64-what-is-the-worst-possible-increase-in-space-usage, Maj 2019.

[20] "Benchmarking personal cloud storage." http://conferences.sigcomm.org/imc/2013/papers/imc092-dragoA.pdf, 2013.

[21] "command line - random image generator - unix & linux stack exchange." https://unix.stackexchange.com/questions/289572/random-image-generator/289670, Maj 2019.

[22] "poisson-process - npm." https://www.npmjs.com/package/poisson-process, April 2019.

[23] "Speedtest by ookla - the global broadband speed test." https://www.speedtest.net, April 2019.

[24] "https://checkip.amazonaws.com/." https://checkip.amazonaws.com/, April 2019.

[25] "Ip echo service." https://ipecho.net/, April 2019.

[26] "Can you provide me with the ip address for my application? - heroku help." https://help.heroku.com/4WADH6LX/can-you-provide-me-with-the-ip-address-for-my-application, April 2019.

[27] "Aws marketplace: Heroku cloud application platform." https://aws.amazon.com/marketplace/pp/Heroku-Inc-Heroku-Cloud-Application-Platform/B008DJG1TY, April 2019.

[28] "Ip location finder - geolocation." https://www.iplocation.net/, April 2019.

[29] "Github - wahengchang/js-meter: it is an tool of measuring performance of time, cpu, ram and heap of javascript code." https://github.com/wahengchang/js-meter, April 2019.

[30] "Heroku labs | heroku dev center." https://devcenter.heroku.com/categories/labs, April 2019.

[31] "Librato - add-ons - heroku elements." https://elements.heroku.com/addons/librato, April 2019.

[32] "Poissonova porazdelitev - wikipedija, prosta enciklopedija." https://sl.wikipedia.org/wiki/Poissonova_porazdelitev, April 2019.

[33] "Poisson distribution - wikipedia." https://en.wikipedia.org/wiki/Poisson_distribution, Maj 2019.

[34] "Node.js." https://nodejs.org/en/. [Dostopano: 25. 3. 2019].

[35] A. Palen, "websocket." https://www.npmjs.com/package/websocket. [Dostopano: 26. 3. 2019].

[36] "ws." https://www.npmjs.com/package/ws. [Dostopano: 26. 3. 2019].

[37] S. Neelakantam, "Implementing a websocket server with node.js." https://hackernoon.com/implementing-a-websocket-server-with-node-js-d9b78ec5ffa8. [Dostopano: 4. 4. 2019].

[38] "Python." https://www.python.org/. [Dostopano: 26. 3. 2019].

[39] "Insertion sort." https://www.geeksforgeeks.org/insertion-sort/. [Dostopano: 29. 3. 2019].

[40] A. Palen, "poisson-process." https://www.npmjs.com/package/poisson-process. [Dostopano: 20. 5. 2019].

[41] "Amazon web services." https://aws.amazon.com/. [Dostopano: 28. 3. 2019].