LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

RUBY ON RAILS TĪMEKĻA LIETOTŅU ATMIŅAS NOPLŪDES

BAKALAURA DARBS

Autors: **Mārtiņš Lapsa**
Studenta apliecības Nr.: ml08134
Darba vadītājs: profesors, Dr. dat. Ģirts Karnītis

RĪGA 2017

ANOTĀCIJA

Pētījumā tiek meklēts iemesls, kāpēc apskatītā Ruby on Rails lietotne pārpilda tai atvēlēto atmiņu produkcijas serverī, par hipotēzi izvirzot to, ka lietotnē ir atmiņas noplūdes. Pētījumā apskatīta tīmeklī pieejamā informācija par Ruby lietotņu atmiņas patēriņa profilēšanu un testēšanu. Praktiskajā daļā tika veikta dažāda veida lietotnes testēšana gan testa, gan produkcijas vidē. Testēšanas rezultāti pētījumā izvirzīto hipotēzi noraidīja, atklājot atmiņas burbuļa veidošanos.

Atslēgas vārdi: Ruby, atmiņas noplūde, atmiņas burbulis.

ABSTRACT

MEMORY LEAKS IN RUBY ON RAILS BASED WEB APPLICATIONS

The aim of research is to find reasons, why a Ruby on Rails based web application requires memory more than web server configuration allows. The hypothesis is that there are memory leaks in discussed web application. Research looks for different techniques available on the web to pinpoint a memory issue cause. Various methods and tools are tried in test and production environment. Hypothesis of paper is denied, as tests reveal memory bloat issues in said web application.

Keywords: Ruby, memory leak, memory bloat.

SATURS

APZĪMĒJUMU SARAKSTS

HTTP	Tīkla Internet standartprotokols, kas nodrošina informācijas apmaiņu globālajā tīmeklī.
HTML	Vispārinātā iezīmēšanas standartvaloda, kuras lietošana ļauj saistīt dokumentus ar izvēlētiem pieejas punktiem.
Pārneses datne	Vieta cietajā diskā, ko izmanto kā datora lasāmatmiņas virtuālo atmiņu.
Produkcijas vide	Datorsistēmas konfigurācija, kurā lietotne pilda savu tiešo uzdevumu un apkalpo gala lietotājus.

IEVADS

Apskatītā tēma šajā pētījumā autoram ir aktuāla darba vietā. Ruby on Rails ražošanas informācijas sistēmas lietotne, kuras izstrādē piedalās autors, patērēja vairāk atmiņas nekā ierasts šāda veida un izmēra lietotnēm. Lietotne dažu stundu darbības laikā aizņēma visu uz servera atvēlēto atmiņu, novedot pie atmiņas deficīta. Līdzšinējais pagaidu risinājums bija servera regulāra, automatizēta restartēšana.

Lietotnes neparasti lielais atmiņas patēriņš par pētījuma hipotēzi liek izvirzīt uzstādījumu, ka lietotnes darbībā ir vērojama atmiņas noplūde.

Darba mērķis ir noskaidrot, vai minētajā lietotnē ir atmiņas noplūdes un kas vainojams pie lielā atmiņas patēriņa, kā arī iezīmēt iespējamos risinājumus.

Pētījumā tiek meklētas metodes, ar kuru palīdzību konstatēt atmiņas noplūdes. Autors veic par šo tēmu tīmeklī atrodamās informācijas izpēti un pielieto iegūto informāciju atmiņas noplūžu meklēšanai, izmantojot dažāda veida testēšanas rīkus un pieejas.

Pirmajā nodaļā ir aplūkota Ruby programmēšanas valodas vispārēji darbības principi un drazu savācēja darbība. Otrajā nodaļā apkopota jaunākā tīmeklī pieejamā informācija nozares profesionāļu emuāros par atmiņas noplūžu meklēšanu Ruby lietotnēs. Trešajā nodaļā uzskaitītas metodes, kas tika pielietotas lietotnes darbības pētīšanai, un to rezultāti.

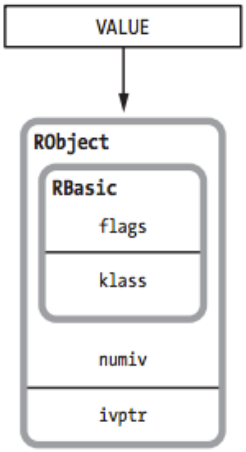
1. RUBY DARBĪBAS PRINCIPI

Lai gan Ruby ir programmēšanas valoda ar vairākām interpretācijām, populārākā un plašāk izmantotā ir C valodas interpretācija MRI (Matz's Ruby Interpeter, nosaukta Ruby radītāja Yukihiro "Matz" Matsumoto vārdā). Atmiņas noplūžu skartā Ruby on Rails lietotne izmanto tieši šo interpretatoru. Turpmāk apskatīsim šī interpretatora darbības principus.

1.1. Vērtību glabāšana atmiņā

Viens no Ruby darbības principiem ir "Jebkas ir objekts". Tas nozīmē, ka jebkāda veida vērtības Ruby uztver un arī glabā kā objektus. Ruby programmas sastāv no objektiem un ziņojumiem starp tiem.

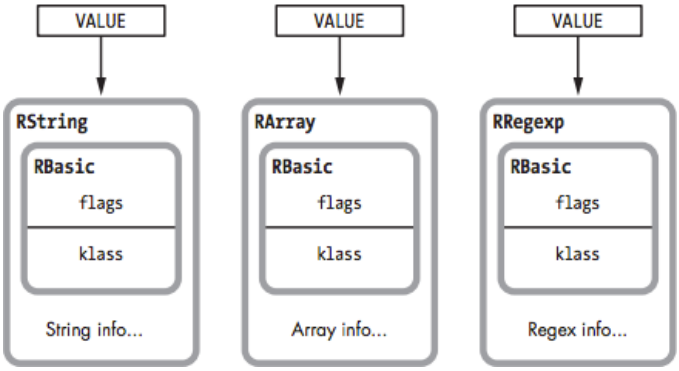
Izveidojot kādu objektu Ruby programmā, tas tiek saglabāts C valodas struktūrā RObject. Uz šīs struktūras atrašanās vietu norāda rādītājs VALUE. Lai norādītu uz kādu vērtību, Ruby iekšēji vienmēr izmanto rādītājus ar šādu nosaukumu. RObject struktūra satur iekšējo struktūru RBasic un informāciju, kas specifiska konkrētajam objektam. RBasic struktūra satur informāciju, ko izmanto visas Ruby vērtības – loģisko vērtību kopu (sauktu par karodziņiem) dažādu iekšējo tehnisko vērtību glabāšanai un klases rādītāju sauktu par *klass*. Klases rādītājs ir piesaistīts klasei, kuras instance ir attiecīgais objekts. Atlikušajā RObject daļā Ruby glabā masīvu ar instances mainīgajiem, ko šis objekts satur. Tam kalpo divas vērtības – *numiv* (number of instance variables) instances mainīgo skaita glabāšanai un *ivptr* (instance variables pointer), kas ir rādītājs uz masīvu, kurā glabājas šie instances mainīgie [1].



1.1. att. **RObject uzbūve**

Augstāk redzamajā piemērā redzama RObject struktūras uzbūve un rādītājs (1.1. att.).

RObject struktūra interpretatora iekšienē paredzēta pielāgotiem jeb tādiem objektiem, kuru vērtība ir kompleksa. Vispārējo tipu objektiem, tādiem kā simbolu virknei, masīvam, veseliem skaitļiem vai peldošā punkta skaitļiem MRI interpretatorā ir paredzētas atsevišķas struktūras. Tā, piemēram, simbolu virknei paredzēta struktūra RString, masīvam – RArray, bet regulārajai izteiksmei – RRegexp (1.2 att.).



1.2. att. **C valodas struktūras vispārējo tipu objektiem**

Dažkārt glabājamā vērtība ir pietiekami maza, ka tās glabāšanai nav vajadzīga atsevišķa struktūra. Mazu veselu skaitļu un vēl dažu vienkāršu vērtību glabāšanai pietiek ar VALUE rādītājam paredzēto vietu. Tādos gadījumos VALUE satur pašu vērtību un tehniskos karodziņus. Piemēram, viens no šādiem karodziņiem ir FIXNUM_FLAG. Ja atbilstošajam bitam vērtība ir *true*, tad Ruby interpretators zina, ka vērtība ir mazs vesels skaitlis, kas ir *Fixnum* klases instance.

Ruby valoda un līdz ar to arī MRI tiek aktīvi attīstīts un uzlabots, bet pamatprincips objektu glabāšanai mainīts netiek. Sekojošajā attēlā redzams, kā visas iepriekš uzskaitītās struktūras ir apvienotas (1.3. att.).

```
typedef struct RVALUE {
    union {
        /*…*/
        struct RBasic  basic;
        struct RObject object;
        struct RClass  klass;
        struct RFloat  flonum;
        struct RString string;
        /* utt.*/
    } as;
} RVALUE;
```

1.3. att. **Dažādo Ruby struktūru apvienojums struktūrā RVALUE**

Iekšēji visus Ruby objektus pārstāv C valodas struktūra RVALUE. MRI izmanto C valodas *union* apvienojumu RVALUE definīcijā, lai iekļautu visas MRI izmantotās struktūras, kā, piemēram, RArray, RString, RRegexp, un tā tālāk [1]. RVALUE struktūras izmērs ir 40

baiti.

Lai šos vienādos, RVALUE struktūrā apvienotos objektus varētu glabāt atmiņā, tie ir kaut kādā veidā jāorganizē. Šim uzdevumam domāta tā sauktā Ruby kaudze. Ruby kaudzes mazākā loģiskā iedaļa ir viens slots. Katrs slots ir tieši tik liels, lai varētu saturēt vienu RVALUE struktūru, un līdz ar to vienu Ruby objektu. Slota lielums tāpat kā RVALUE struktūra ir 40 baiti.

Protams, ka 40 baitos ne vienmēr var saglabāt visu objekta informāciju. Tāpēc visa informācija, ko objekts aizņem atmiņā, netiek glabāta slotā. Katrs slots ir maza, fiksēta izmēra vieta, kuru var uztvert kā Ruby interpretatora atmiņas atrašanās vietas turi. Šī atrašanās vieta atrodas ārpus pašas Ruby kaudzes un tā satur pašus objekta datus. Lai būtu skaidrs – ja mums ir 50 MB liela simbolu virkne, tā netiek glabāta Ruby kaudzē. Vieta šai simbolu virknei tiek piešķirta ar kaut ko līdzīgu *malloc* komandai C valodā un tad šīs simbolu virknes dati tiek saglabāti piešķirtajā vietā Sistēmas kaudzē. Slots Ruby kaudzē vienkārši satur referenci uz šo atmiņas atrašanās vietu Sistēmas kaudzē, kas satur šīs simbolu virknes 50 MB datus [2].

Tātad, tiklīdz mēs Ruby kodā izveidojam jaunu objektu (1.4. att.), Ruby kaudzē kādā no brīvajiem slotiem tiek ievietota jau RVALUE struktūra.

```
foobar = MyFoobar.new
```

1.4. att. **Jauna objekta izveidošana Ruby kodā**

Parasti Ruby kaudzē ir atrodami brīvi sloti, kuros ievietot jaunus objektus. Iespēja, ka Ruby kaudzē būs kāds brīvs slots mūsu jaunizveidotajam objektam, ir diezgan augsta. Ruby pārvalda atmiņu tā, lai tam nebūtu kodolam (*kernel*) jāprasa papildu atmiņa katru reizi, kad tiek izveidota jauna objekta instance [3].

Tam par iemeslu ir tā sauktās Ruby kaudzes lapas. Katra lapa ir 16 KB liela un tātad var saturēt aptuveni 408 objektu slotus. Startējot lietotni, sākumā tiek izveidota kaudze ar 25 lapām.

Bet kas notiek, ja Ruby kaudzē vairs nav neviena lapa ar brīvu slotu? Tādā gadījumā iesaistās Ruby drazu savācējs.

1.2. Ruby drazu savākšana

Pretēji nosaukumam drazu savākšana nav tikai atmiņas atbrīvošana no vairs nevajadzīgiem objektiem. Patiesībā drazu savācēji risina trīs problēmas:

- tie piešķir atmiņu jauniem objektiem;
- tie identificē objektus, ko lietotne vairs neizmanto;
- tie atgūst atmiņu no vairs neizmantotiem objektiem.

Ruby drazu savākšanas sistēma arī pilda šīs funkcijas. Kad izveido jaunu Ruby objektu,

drazu savācējs šim objektam piešķir atmiņu. Vēlāk drazu savācējs nosaka, kad programma šo objektu ir beigusi izmantot, lai varētu šo pašu atmiņu izmantot citu objektu izveidošanai. Atmiņas piešķiršana un atgūšana ir monētas divas puses, tāpēc ir loģiski, ka Ruby drazu savācējs izpilda abas šīs darbības [1]. Tieši tāpēc RVALUE struktūra Ruby valodas kodā ir nodefinēta tajā pašā datnē, kur ir definēts drazu savācējs.

Kopš 2.1 versijas Ruby izmanto paaudžu drazu savācēju ar "atzīmē un iztīri" (*mark-and-sweep*) pieeju. Pašreizējo Ruby drazu savācēja implementāciju sauc par RGenGC, un to izstrādāja Koichi Sasada Ruby kodola komandas sastāvā.

Vienkāršoti runājot, "atzīmē un iztīri" drazu savācējs pārvietojas pa Ruby kaudzi (kas būtībā ir objektu grafs) un atzīmē, kuri objekti joprojām tiek izmantoti, un kuri nē. Neizmantotie objekti tiek dzēsti, atbrīvojot atmiņu jauniem objektiem. Izmantotie objekti paliek neskarti.

Sākot ar objektu, par kuru zināms, ka tas ir referencēts programmā, drazu savācējs seko katrai referencei no šī objekta uz citiem objektiem kaudzē. Katru reizi, kad drazu savācējs atrod objektu, ir skaidrs, ka šis objekts vēl joprojām tiek izmantots, jo to referencē objekts, kas tika aplūkots pirms tam. Drazu savācējs atzīmē šo objektu, ieslēdzot vienu no bitu karodziņiem un turpina referenču grafa apstaigāšanu, līdz vairs nevar atrast nevienu norādi.

Pēc atzīmēšanas fāzes sākas tīrīšanas fāze. Drazu savācējs iet cauri visai kaudzei un iztīra visus neizmantotos objektus, kuriem pirms tam nebija pacēlis karodziņu. Atzīmētajiem objektiem karodziņš tiek nolaists, tādējādi sagatavojoties jaunai drazu savākšanas iterācijai.

Paaudžu drazu savācējs, kas izmanto "atzīmē un iztīri" pieeju, būtībā strādā tā pat, bet izmanto dažus papildinājumus ātrākai objektu apstaigāšanai.

Paaudžu drazu savācējs izmanto pieņēmumu, ka lielākā daļa objektu kļūst nevajadzīgi, kamēr tie ir pavisam jauni. Tātad jauni objekti, kas izveidoti kopš iepriekšējās drazu tīrīšanas operācijas, visdrīzāk referencēs vecus objektus, kas ir pārdzīvojuši iepriekšējo drazu tīrīšanu. Tikai neliela daļa jauno objektu būs jāatzīmē un jāpasaudzē, un tie būs tie objekti, kurus referencēs vecie objekti.

Balstoties uz šo pieņēmumu, drazu savācējs var ietaupīt daudz laika un nedarīt nevajadzīgu darbu, ja tas koncentrējas uz jauniem objektiem, jo tie ir tie, kas visdrīzāk būs jāatbrīvo. Apstaigājot vecos objektus, netiktu atbrīvots tik daudz atmiņas, kā apstaigājot jaunos objektus.

Tas nozīmē, ka paaudžu drazu savācējam ir vajadzīgi divi darbības režīmi. Ruby gadījumā tos sauc par mazo (*minor*) un lielo (*major*). Mazajā režīmā drazu savācējs apstaigā tikai jaunos objektus, bet lielajā režīmā tas apstaigā visu objektu grafu, ieskaitot veco objektu paaudzi. Mazajam režīmam būtu jābūt daudz ātrākam un parasti drazu savācējs mazajā režīmā notiek daudz biežāk nekā lielajā režīmā.

Lai varētu klasificēt objektus jaunajos un vecajos, drazu savācējs rīkojas sekojoši. Kad atzīmēšanas fāzē tiek atzīmēts kāds objekts (kas nozīmē, ka objekts pārdzīvos drazu savākšanu), tas tiek pieskaitīts vecajai paaudzei. Neatzīmētie objekti tiek iztīrīti. Nākamajā drazu savākšanas reizē vecā paaudze var tikt ignorēta.

Bet ir viena problēma. Ja vecās paaudzes objekts sāk referencēt jaunu objektu starp drazu savākšanas reizēm, nākamajā reizē drazu savācējs neatzīmēs šo jaunizveidoto objektu, jo vecās paaudzes referencējošais objekts ar savām referencēm tiks ignorēts.

Šīs problēmas novēršanai vajadzīga atcerēšanās kopa (*remember-set*) un rakstīšanas barjera (*write-barrier*). Apkārt katram objektam tiek uzlikta rakstīšanas barjera, kura jāpārvar katrai objekta rakstīšanas darbībai. Svarīgi, ka arī references norādīšana uz objektu ir rakstīšanas darbība.

Piemērs no C valodas koda, kas ilustrē references pievienošanu, kas ir rakstīšanas darbība (1.5. att.). "&new_object" norāda uz vietu atmiņā, kurā glabājas "new_object" objekts. Piemēra rindiņa ir instrukcija, lai masīva pirmajā vietā ievieto šo atmiņas vietas adresi.

```
old_array[0] = &new_object
```

1.5. att. **References norādīšana valodā C**

Ar rakstīšanas barjeru ap objektiem drazu savācējs var identificēt references no veciem objektiem uz tikko izveidotiem objektiem. Kad tas notiek, tas pievieno referenci uz veco objektu atcerēšanās kopā.

Bez atcerēšanās kopas drazu savācējs pārlēktu vecās paaudzes objektus mazajā režīmā un nevarētu atzīmēt jaunos objektus, ko šie vecie objekti referencē. Bet ar rakstīšanas barjeru un atcerēšanās kopu drazu savācējs var gan apstaigāt jaunās paaudzes objektus, gan apstaigāt tos vecās paaudzes objektus, kas iegaumēti atcerēšanās kopā, un līdz ar to neizlaist tos jaunos objektus, uz kuriem norāda references no vecajiem objektiem.

Ruby objekti tiek klasificēti gaišajos (*sunny*) un tumšajos (*shady*). Gaišie objekti ir tie, kurus aizsargā rakstīšanas barjera, bet tumšajiem barjeras nav.

Ņemot vērā šo iedalījumu, mazais drazu savācēja RGenGC režīms darbojas mazliet sarežģītāk. Kad drazu savācējs apstaigāšanas laikā atrod tumšu objektu, tas paceļ objekta karodziņu, bet neieceļ šo objektu vecajā paaudzē. Tam par iemeslu ir tumšā objekta rakstīšanas barjeras neesamība. Tumša objekta iecelšana vecajā paaudzē rezultētos pazudušās referencēs no veciem objektiem uz jauniem. Bet tumšo objektu mērķis ir nepalaist garām references. Tāpēc drazu savācējs pārbauda, vai kāds vecais objekts referencē tumšo objektu, un, ja tā, tad ieceļ tumšo objektu atcerēšanās kopā.

Objekts, kas izveidots kā gaišais, tāds var nebūt visu savu pastāvēšanas laiku. Drazu savācējs var aptumšot objektus. Objektu aptumšošana nozīmē objekta pārcelšanu no gaišā uz

tumšo, izņemšanu no vecās paaudzes un pievienošanu atcerēšanās kopai. Tas notiek tad, ja atmiņas adresei notiek zema līmeņa piekļuve caur C lietojumprogrammu saskarni, ko Ruby virtuālā mašīna var noteikt. Pēc tam, kad C lietujumprogrammu saskarnes lietotājs ir ieguvis referenci uz objektu, efektīva rakstīšanas barjera vairs nav iespējama, tāpēc objekts tiek aptumšots.

Tā vietā, lai RGenGC drazu savācējs vienkārši paļautos uz rakstīšanas barjerām referenču noteikšanai, tas pievieno tumšos objektus atcerēšanās kopai, kad tas nevar pateikt, vai šis objekts referencē jaunu objektu.

Nakamajā darbības reizē drazu savācējs var apstaigāt jaunās paaudzes objektus, kā arī tos objektus, kas ir atcerēšanās kopā. Šī kopa satur vecos objektus, kam ir references uz jauniem objektiem un tumšos objektus, kurus referencē vecie objekti. Apstaigājot šo kopu, drazu savācējs nepalaidīs garām atzīmēšanai tos objektus, kas ir tikko izveidoti.

Ruby kaudzes lapas ir sadalītas divās kategorijās – *eden* un *tomb*. Ja lapai ir kaut viens aizņemts slots, tā ir *eden* lapa. *Tomb* lapās savukārt nav neviena aizņemta slota. Tīrīšanas fāzē drazu savācējs, meklējot objektus dzēšanai, skatās tikai *eden* lapas un iztīra nevajadzīgos objektus no tām. Šis process notiek pakāpeniski pa vienai *eden* lapai. Ja pēc šīs tīrīšanas vairs nav palicis neviens objekts, lapa tiek ielikta *tomb* kategorijā. Šī darbība palīdz samazināt objektu fragmentāciju pa lapām, jo jauni objekti sākumā tiek likti tikai *eden* lapās. Tikai tad, kad *eden* lapās vairs nav brīvu slotu, kāda no *tomb* lapām tiek pievienota atpakaļ pie *eden* lapām, lai tajā varētu ievietot objektus.

Ja drazu savākšanas laikā neizdodas atbrīvot pietiekami daudz vietas Ruby kaudzē, tai ir jāpievieno jaunas lapas.

1.3. Atmiņas noplūdes

Atmiņas noplūdes, tas ir, programmai piešķirta atmiņa, ko tā vairs neizmanto, palēnina programmas izpildi, palielinot lapošanu, un var izraisīt programmai pieejamās atmiņas izbeigšanos. Atmiņas noplūdes ir daudz grūtāk noteikt nekā neatļautu pieslēgšanos atmiņai. Atmiņas noplūdes notiek, kad atmiņas apgabals netiek atbrīvots, un tātad ir bezdarbības nevis darbības rezultāts. Atmiņas noplūdes reti rezultējas tieši novērojamās kļūdās, bet tā vietā kumulatīvi samazina vispārējo veiktspēju [4].

Noprotams, ka Ruby gadījumā atmiņas noplūdes varētu rasties gadījumos, ja drazu savācējs no Ruby kaudzes neiztīra nevajadzīgu objektu. Tomēr, kā minēts iepriekš, Ruby drazu savācējs rūpīgi pārbauda visus objektus un to, vai tiem ir references. Tas nozīmē, ka klasiskas atmiņas noplūdes, pie kurām varētu vainot Ruby drazu savācēju, ir maz iespējamas.

Bet tas nenozīmē, ka lietotnes ir pasargātas no atmiņas noplūdēm. Viens no iemesliem, kāpēc var veidoties atmiņas noplūdes, ir plašais klāsts ar atvērtā koda Ruby pakotnēm. Tās satur dažādas noderīgas bibliotēkas, moduļus vai veselas lietotnes, ko var izmantot lietotnes izstrādē. Tipiska Ruby on Rails lietotne satur vairākus desmitus dažādu pakotņu. Lai gan parasti šīs pakotnes satur Ruby valodā rakstītu kodu, citas optimizācijas nolūkos ir rakstītas uzreiz C valodā. Tā kā Ruby drazu savācējs neatbild par šī koda izpildi, kodam jābūt rakstītam, iekļaujot pašrocīgu drazu savākšanu. Bet tam nav nekādas garantijas. Līdz ar to izstrādātājam jābūt uzmanīgam ar šādu pakotņu izmantošanu.

Cits iemesls pakāpeniskai atmiņas aizpildei var būt kāds objekts, kas lietotnes darbības laikā tiek izmantots un pamatoti netiek dzēsts, bet kā saturs tiek regulāri papildināts. Piemēram, tas var būt kāds masīvs, kuram visu lietotnes darbības laiku tiek pievienoti jauni elementi.

Viena no vēsturiskajām atmiņas noplūžu kļūdām bija Ruby drazu savācēja īpatnība, ka tas nedzēsa Ruby simbola objektus. Simbola objekts Ruby valodā ir simbolu virkne ar tādu īpatnību, ka tā Ruby kaudzē tiek izveidota vienreiz. Ja pēc tam tiek izveidots jauns mainīgais, kura vērtība ir norādīta kā šis simbols, šim mainīgajam norāde uz vērtību norāda uz jau iepriekš izveidoto simbolu virkni. Tas noder, lai neveidotu bieži atkārtojošas simbolu virknes dažādos lietotnes kontekstos [5]. Sākotnējo Ruby versiju drazu savācēji šos simbolus netīrīja, jo bija uzskats, ka tos lietotnes darbības turpinājumā tik un tā kaut kad vajadzēs. Tomēr tas pavēra iespēju lietotnē uzkrāties bazgalīgi lielam daudzumam simbolu un līdz ar to atmiņas noplūdēm. Pastāvēja pat tāds apzīmējums kā “simbolu pakalpojuma atteikums”. Zinot, ka lietotne no padotajiem datiem veido simbolus un veidojot pieprasījumus ar dažādiem datiem, varēja panākt, ka lietotnē aizpildās atmiņa un lietotnes darbība pārtrūkst [6]. Tomēr ar Ruby 2.2 versiju šī kļūda tika labota. Simbolu darbības princips nemainījās, bet tie tiek dzēsti no Ruby kaudzes, ja uz tiem nav nevienas norādes.

Izpētot par Ruby darbību pieejamo informāciju, jāsecina, ka īstas atmiņas noplūdes lietotnēs ir maz iespējamas. Ruby valoda ir labi uzturēta un atjaunināta, tās drazu savācēja darbība ir krietni uzlabota salīdzinājumā ar sākotnējām versijām.

2. CITU PIEREDZE

Ruby programmētāju vidū problēmas ar pārāk lielu atmiņas patēriņu nav reta parādība. Tīmeklī ir atrodami daudz un dažādi raksti par šo tēmu. Tiesa, jāņem vērā iepriekš minētais, ka Ruby nemitīgi attīstās, atmiņas menedžments laika gaitā ir mainījies un daļa no rakstiem ir zaudējuši savu aktualitāti.

Viens no vairāku rakstu autoriem par šo tēmu ir Ričards Šnēmans (*Richard Schneeman*). Viņš ir strādājis gan pie Ruby valodas, gan pie Ruby on Rails ietvara. Šobrīd viņš strādā kompānijā Heroku, kas nodarbojas ar lietotņu uzturēšanu mākoņpakalpojumu veidā. Savā rakstā "Debugging a Memory Leak on Heroku" viņš apraksta svarīgākās lietas, kas būtu jāzina Ruby programmētājam saistībā ar atmiņas lietojumu.

Šnēmans uzsver, ka, visticamāk, atmiņas noplūde lietotnē nav. Ruby valoda atmiņu pieprasa dinamiski, un ir tikai normāli, ja pēc lietotnes startēšanas tā aizņem arvien vairāk un vairāk atmiņas. Iemesls, kāpēc programmētājiem parasti ir aizdomas par atmiņas noplūdēm, ir tāds, ka viņi novēro atmiņas patēriņu pārāk mazu laika sprīdi. Viņš uzsver, ka lielākajai daļai lietotņu atmiņas patēriņam pēc kāda laika būtu jāizlīdzinās, tas ir, lietotne vairs nepieprasīs atmiņu un spēs darboties ar to atmiņas daudzumu, kuru jau pieprasījusi. Ja lietotne aizņem daudz vairāk atmiņas nekā tai vajadzētu, bet laika gaitā tā neaizņem arvien vairāk un vairāk atmiņas, tas nozīmē, ka lietotne neefektīvi rīkojas ar tai piešķirto atmiņu un veidojas tā sauktais atmiņas burbulis [7].

Tālāk rakstā Ričards Šnēmans dod dažādus padomus, kā samazināt lietotnes atmiņas patēriņu. Kā pirmo viņš iesaka izmēģināt laiksakritīgo procesu skaita samazināšanu. Tīmekļa serveriem ir jāspēj apstrādāt vairākus pieprasījumus vienlaicīgi, tāpēc tiek veidoti vairāki procesi. Lai arī šie procesi spēj izmantot vienu un to pašu atmiņu un tās objektus, katrs no tiem tik un tā aizņems papildu vietu atmiņā. Ruby lietotnes visbiežāk izmanto Puma tīmekļa serveri. Tam konfigurācijas uzstādījumos samazinot šo procesu skaitu, var panākt aizņemtās atmiņas samazināšanos. Negatīvā puse šai darbībai ir tāda, ka samazināsies tīmekļa servera spēja atbildēt uz pieprasījumiem.

Nākamais ieteikums no Šnēmana ir lietot viņa izstrādāto Puma Worker Killer moduli [8]. Tas periodiski restartē Puma tīmekļa servera procesus, tādējādi atgriežot lietotni sākuma stāvoklī un samazinot atmiņas patēriņu tajā brīdī. Tomēr viņš uzsver, ka tas nav ilgtspējīgs risinājums un nenovērš pašu problēmu, bet vienkārši cīnās ar sekām.

Vēl Ričards Šnēmans iesaka pārliecināties, vai lietotnes izmantoto moduļu vidū nav kāds, kas aizņem pārāk daudz atmiņas ielādes brīdī. To var izdarīt ar Šnēmana izveidoto rīku "Derailed benchmarks" [9]. Izmantojot šo rīku, var redzēt, cik atmiņas aizņem katrs no

pieprasītajiem moduļiem lietotnes palaišanas brīdī. Palaižot šo rīku, tas izanalizē un izvada visus lietotnē izmantotos moduļus un tiem vajadzīgo atmiņas izmēru (2.1. att.).

```
$ bundle exec derailed bundle:mem
TOP: 54.1836 MiB
  mail: 18.9688 MiB
    mime/types: 17.4453 MiB
    mail/field: 0.4023 MiB
    mail/message: 0.3906 MiB
  action_view/view_paths: 0.4453 MiB
    action_view/base: 0.4336 MiB
```

2.1. att. **Derailed benchmarks rīka darbības izvades piemērs**

"Derailed benchmarks" rīks moduļu atmiņas analīzes režīmā izvada hierarhisku sarakstu, kura pirmajā rindā redzams kopējais visu ielādējamo moduļu atmiņas apjoms. Piemērā redzams, ka no kopējiem 54 megabaitiem, *mail* modulis ar tam vajadzīgajiem apakšmoduļiem aizņem gandrīz 19 megabaitus, savukārt *action_view/view_paths* aizņem tikai gandrīz pusi megabaita.

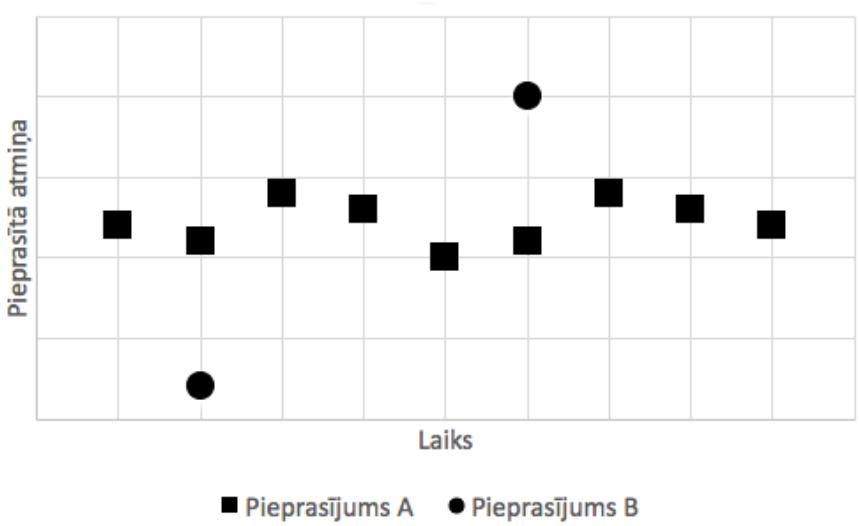
Šnēmans iesaka pārbaudīt šo sarakstu un izņemt nevajadzīgos moduļus, kuri, iespējams, ieviesti izstrādes gaitā, kā arī mēģināt atjaunināt tos moduļus, kuri ir vajadzīgi lietotnē.

Tikai tālāk savā rakstā Šnēmans pievēršas pārāk lielam objeku daudzumam lietotnes darbības brīdī. Viņš min iespēju izmantot attālinātos profilēšanas rīkus, bet tie pārsvarā ir par maksu. Tāpēc var izmantot to pašu "Derailed benchmarks" moduli, ar ko var noskaidrot, cik, kur un kāda veida objektus izveido viens vai vairāki pieprasījumi lietotnei. Iedarbinot šo rīku attiecīgajā režīmā, tas iedarbina lietotni un veic pieprasījumu vienu vai vairākas reizes un izvada lietotnes datņu sarakstu, kas lietotnes darbības laikā pieprasa vislielāko atmiņas apjomu. Šnēmans iesaka sākt testēšanu ar vislielākajiem pieprasījumiem, uz kuriem lietotne atbild visilgāk.

Tikai pašās beigās Šnēmans pievēršas atmiņas noplūdēm un pieļauj, ka arī tādas var rasties. To testēšanai viņš "Derailed benchmarks" rīkā ir izveidojis funkcionalitāti, kas testē vienu konkrētu, norādāmu pieprasījumu, pieprasot to vairākus simtus reižu, un ik pa piecām sekundēm izvada lietotnes izmantotās atmiņas daudzumu. Ja izmantotās atmiņas daudzums tikai palielinās, tad var izdarīt secinājumu, ka attiecīgajā lietotnes vietā tiešām ir atmiņas noplūde.

Viens no attālinātās monitorēšanas servisiem Scout ir savā tīmekļa vietnē ir publicējis rakstu par atmiņas burbuļiem [10]. Arī šis raksts uzsver atšķirību starp atmiņas burbuļiem un noplūdēm. Atmiņas burbulis izveidojas īsā laikā, pieprasot daudz atmiņas. Savukārt atmiņas noplūdes var būt mazas un nemaz tik viegli pamanāmas, tāpēc sākotnēji vajadzētu koncentrēties uz atmiņas burbuļu meklēšanu. Lai arī Ruby atbrīvo atmiņu, ja tā tam ir nevajadzīga, tas to dara

ļoti lēni un negribīgi, jo pilnīgi iespējams, ka nākamajā pieprasījumā atkal būs vajadzība pieprasīt atmiņu. Tāpēc labāk ir uzskatīt, ka Ruby atmiņu operētājsistēmai neatgriež. Ja reiz tas ir pieprasījis noteiktu atmiņas daudzumu, tad to neatdos. Bet ir iespēja, ka turpinājumā pieprasīs vēl vairāk.



2.2. att. **Pieprasījumi un to pieprasītā atmiņa**

Ja veiktspējas optimizācijā mēs pievērstu uzmanību pieprasījumam A, jo tas notiek bieži un tam ir vajadzīgs zināms daudzums atmiņas, tad atmiņas burbuļu novēršanā ir jāpievērš uzmanība pieprasījumam B, jo, lai arī tas notiek daudz retāk, pietiek ar vienu reizi, lai tas uzpūstu vajadzīgo atmiņas daudzumu, un mēs pieņemam, ka šī atmiņa nekad netiks atdota atpakaļ (2.2. att.).

Scout rakstā uzsver, ka uzmanību jāpievērš lietotājiem ar plašām pieejas tiesībām. Ja ierindas lietotājs redzēs saturu, kas attiecas tikai uz viņu, tad lietotājs ar plašām tiesībām redzēs visu lietotnes saturu. Liela datu apjoma ielāde bieži vien ir atmiņas burbuļu cēlonis.

Tāpat šajā rakstā norādīts, ka jākoncentrējas uz maksimālo iespējamo atmiņas piešķiršanu Ruby on Rails lietotnes kontroliera darbībā. Tas būtībā ir gadījums ar lietotājiem, kuriem ir plašas pieejas tiesības.

Laba ir norāde, ka lietotnē, kura darbojas ilgstoši, vairs nevar būt korelācija starp Ruby kaudzes izmēru un patērēto atmiņu. Fakts, ka kāda izsaukuma laikā palielinās kaudzes izmērs vēl nenozīmē, ka šiem objektiem būs vajadzīga papildu atmiņa. Lielo atmiņas daudzumu var būt pieprasījusi kāda cita darbība pirms tam. Tāpēc jāņem vērā, ka uzreiz pēc lietotnes startēšanas patērētās atmiņas daudzums augs strauji, kamēr lietotnē nebūs dinamiski ielādēta lielākā daļa vajadzīgo moduļu.

Raksta turpinājumā tiek uzskaitītas biežākās Ruby on Rails atmiņas problēmas. Parasti visvairāk atmiņas piešķiršana notiek nevis veidojot ActiveRecord objektus, bet gan pēc tam tos atveidojot skatos. Tiek izveidotas ļoti daudzas simbolu virknes, lai varētu atveidot objekta

daudzos atribūtus, kā arī apkārt tiem neieciešamo HTML kodu no šablona, piemēram, tabulas šūnas un citus elementus. Tāpēc svarīgi sarakstu skatos izmantot lapošanu.

Kā nākamā uzrādīta ActiveRecord moduļa N+1 pieprasījumu problēma datu bāzei. N+1 nozīmē pieprasījumu skaitu datu bāzei, kur N ir objektu skaits. Ja vēlamies iegūt, piemēram, sarakstu ar pasūtījumiem, pie reizes iekļaujot arī klienta nosaukumu, kas glabājas saistītajā klientu sarakstā, ir diezgan vienkārši panākt N+1 pieprasījumu, ja pie sākotnējās pasūtījumu ielādes nenorādām, ka datu bāzē nepieciešams veikt JOIN operāciju, lai uzreiz iegūtu arī datus par katra pasūtījuma klientu. Intuitīvāk šo problēmu būt saukt par 1+N problēmu. Ja veicam JOIN operāciju, ir nepieciešams tikai viens pieprasījums datu bāzei. Savukārt, to nedarot, mums jāveic šis pirmais pieprasījums, lai iegūtu pasūtījumu sarakstu, un tad katram pasūtījumam viens pieprasījums klienta datu iegūšanai. N+1 pieprasījumi ne tikai palielina servera atbildes laiku, bet izraisa arī aptuveni divas reizes vairāk objektu izveidošanu.

Veicot pieprasījumus datu bāzei, ir izšķērdīgi ielādēt visas tabulas kolonnas, ja nepieciešamas tikai dažas. Pirmkārt, vairāk datu saņemšanai no datu bāzes nepieciešams vairāk laika, otrkārt, šie dati ievērojami palielina lietotnes darbībai nepieciešamo atmiņu. Ja kādai tabulai ir kolonna, kurā dati aizņem daudz vietas, ieteicams tiem izveidot atsevišķu tabulu. Tādā veidā šie dati tiks ielādēti tikai tad, kad tas patiešām būs nepieciešams.

Austrālijas nekustamo īpašumu tirdzniecības uzņēmuma "realestate.com.au" tehniskās nodaļas darbinieks Luiss Simono (Louis Simoneau) ir aprakstījis savu pieredzi, meklējot atmiņas noplūdes. Viņš raksta, ka lietotne sākusi patērēt daudz vairāk atmiņas kopš iepriekšējās lietotnes atjauninašanas. Tā kā atjauninājumi ir bijuši visai lieli, viņi nav spējuši novērtēt visai lielās izmaiņas, lai skatoties kodā atrastu lielo atmiņas patērētāju [11].

Lai atrastu vainīgās izmaiņas, Simono izmantoja ObjectSpace klasi, kas pieejama no Ruby 2.1 versijas. Tā dod iespēju piefiksēt visu objektu izveidošanu un to, kura vieta kodā par to atbildīga. Šī darbība izveido žurnālu ar milzīgu informācijas apjomu un krietni paildzina katru darbību, tāpēc nav ieteicams to izmantot produkcijas vidē.

```
class PropertiesController
  def dump
    io = File.open("/tmp/heap_#{params[:id]}", "w")
    ObjectSpace.dump_all(output: io)
    io.close
  end
end
```

2.3. att. **Lietotnes kontrolieris objektu izveides žurnalēšanai**

Simono iespējoja žurnalēšanu un izveidoja atsevišķu kontrolieri lietotnē, kas bija atbildīgs par šīs informācijas ievietošanu atsevišķā datnē (2.3. att.).

Pēc tam viņš izmantoja produkcijas vides datu bāzē esošās ģenerētos URL vietrāžus,

automatizēti veidojot pieprasījumus lietotnei. To laikā tika veikti trīs objektu žurnāla uzņēmumi ar 10 minūšu intervālu. Tas vajadzīgs, lai varētu identificēt tos objektus, kas ir izveidoti laikā starp pirmo un otro uzņēmumu, bet vēl joprojām nav dzēsti, kad veikts trešais uzņēmums.

Analizējot pēdējo uzņēmumu, tik pamanīta objektu izveide Sprockets modulī, kas parasti domāts attēlu, JavaScript un stila lapu sagatavošanai, lai serveris tos viegli varētu izmantot. Šis process notiek lietotnes inicializācijas posmā. Tomēr šajā lietotnē šis modulis tikai izmantots arī citām vajadzībām. Rezultātā modulis kešatmiņā lika arvien jaunas un jaunas vērtības. Lai arī tā nav atmiņas noplūde klasiskā izpratnē, tomēr to par tādu var uzskatīt, jo atmiņa ir aizņemta ar nevajadzīgiem datiem, kas netiek izmantoti.

Rakstā, ko publicējis Bruzs Marzolfs, ārštata Ruby programmētājs, arī minētas iepriekš uzskaitītas lietas. Izmantojot koda piemēru, kurā veido 1 000 000 elementus lielu masīvu, Marzolfs pierāda, ka sākotnēji šķietama atmiņas noplūde patiesībā ir atmiņas burbulis. Viņš norāda, ka ar atmiņas burbuļiem var cīnīties, pielietojot "skaldi un valdi principu" un sadalot uzdevumu mazākās daļās [12]. Tālāk savā rakstā viņš pievēršas dažiem koda fragmentiem, un demonstrē, kā var atrast vainīgās koda vietas, izmantojot atmiņas profilēšanas rīku "memory_profiler". Diemžēl viņš nenorāda, kā pirms tam ir lokalizējis problēmu konkrētajā koda gabalā.

Rakstā, ko publicējis Kallums Draidens (Callum Dryden), ieteikts izmantot profilēšanas rīku "ruby-prof", kas izmantojams kā ielāps Ruby interpretatoram. Draidens ar koda piemēriem rāda, kā pielietot profilēšanas rīku, kā arī to, kā interpretēt iegūtos rezultātus. Tomēr arī Draidens neapraksta savu testēšanas stratēģiju, vai viņš aplūko katru lietotnes funkciju atsevišķi, vai pa kādiem kopumiem. Raksta beigās Draidens min, ka izmanto New Relic monitoringa rīka maksas pakalpojumus, kas norāda uz darbībām, kas pieprasa visvairāk atmiņas [13].

Tīmeklī pieejama iniciatīva, kuras mērķis ir uzskaitīt Ruby moduļus ar zināmām atmiņas burbuļu vai noplūžu problēmām [14]. Tajā norādīti vairāki plaši zināmi Ruby moduļi un to versijas, par kuriem zināms, ka tiem ir problēmas ar atmiņas noplūdēm vai burbuļiem. Pēdējo reizi saraksts atjaunināts 2016. gadā, tomēr lielākā daļa uzskaitīto moduļu versijas vairs nav aktuālas.

Apkopojot tīmekļa emuāros pieejamo informāciju par atmiņas problēmām Ruby lietotnēs, apstiprinās iepriekšējā nodaļā secinātais, ka īstas atmiņas noplūdes ir maz iespējams. Vairāki autori uzsver, ka ieteicams koncentrēties uz atmiņas burbuļiem nevis noplūdēm. Tiek aprakstīta vairāku profilēšanas rīku izmantošana, tomēr ļoti maz informācijas ir par to, kā lokalizēt funkcionalitāti, kas rada pārlieku lielu atmiņas aizpildīšanos.

3. ATMIŅAS NOPLŪŽU MEKLĒŠANA INFORMĀCIJAS SISTĒMĀ

3.1. Pētāmās lietotnes apraksts

Atmiņas pārpildīšanās problēmu skartā lietotne ir ražošanas procesu vadības informācijas sistēma. Tā nodrošina vairāku virzienu funkcionalitāti.

Viena no iespējam ir definēt ražošanas produktu, tā konfigurācijas iespējas, kā arī konfigurācijai atbilstošās detaļas un to nepieciešamo izmēru rēķināšanas formulas. Produkcijas piedāvājums nemainās bieži, tāpēc šo funkcionalitāti uzņēmumā izmanto salīdzinoši reti.

Visbiežāk sistēmu izmanto, lai tajā ievadītu pasūtījumus, norādītu izvēlētās produkcijas konfigurāciju un izmērus. Šī funkcionalitāte ir ne tikai visbiežāk izmantotā, bet arī prasa visvairāk sistēmas resursus. Norādot ražojamā produkta konfigurāciju un izmērus, pēc katras izmaiņas notiek kalkulācijas, kas nosaka ne tikai to, kādi parametri vēl ir vajadzīgi pilnam aprēķinam un vai parametri vispār ir korekti, bet arī izmantojamo detaļu vajadzīgos izmērus un atbilstošo cenu, kā arī ražošanas cehā veicamās darbības, to ilgumu un atbilstošo cenu.

Pasūtījumu veikšana iet roku rokā ar citu funkcionalitāti – ražošanas plānošanu. Pasūtījuma ražošanai nepieciešamās operācijas nonāk ražošanas plānotājā, kur tās var secīgi saplānot pa ražošanas ceha stacijām konkrētās dienās.

Šī sistēma veic arī dažādu ar pasūtījumiem un ražošanu saistītu atskaišu ģenerēšanu gan izklājlapu, gan PDF formātā. Piemēram, katra pasūtījuma dzīves ciklā tiek veikta rēķina ģenerēšana. Savukārt ražošanas ceha darbinieki var izdrukāt attiecīgajai dienai veicamo darbību plānu.

Šī informācijas sistēma ir cieši saistīta ar grāmatvedības informācijas sistēmu Horizon. Izmantojot Horizon XML lietojumprogrammu saskarni, sistēma sinhronizē datus par pasūtījumiem, klientiem, maksājumiem, ražošanas detaļām un citu ar to saistītu informāciju. Šie darbi ir realizēti kā fona darbs. Daļa no tiem notiek regulāri pēc grafika, bet citi – pēc pieprasījuma.

Kā pēdējais darbības virziens jāpiemin informācijas sistēmas lietojumprogrammu saskarne darbam ar trešās puses izstrādātu tīmekļa veikalu. Caur šo saskarni informācijas sistēma nodod informāciju par pieejamiem iepriekš nokonfigurētiem produktiem un to cenām, bet pieņem informāciju par izdarītajiem pasūtījumiem caur šo tīmekļa veikalu.

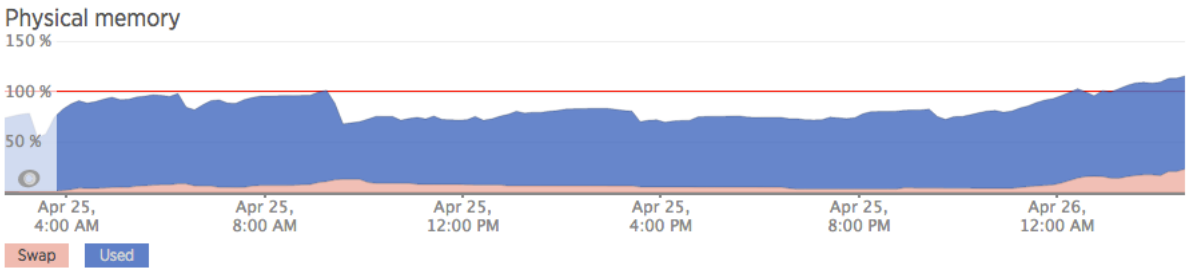
Informācijas sistēmas produkcijas vide ir izvietota uz Amazon Web Services mākoņskaitļošanas pakalpojumu serveriem. Virtuālo serveri darbina Ubuntu 14.14 versijas operētājsistēma. Tas darbina divas Puma tīmekļa serveru instances paralēlai HTTP

pieprasījumu apstrādei, kā arī vienu instanci fona darbu apstrādei [15]. Virtuālajam serverim ir 3,75 GB atmiņas ierobežojums. Šādu atmiņas ierobežojumu lietotnes izstrādāja uzņēmumā izmanto pārsvarā visām izstrādātajām lietotnēm.

Lietotnes darbības monitoringam tiek izmantots "New Relic" tīmekļa pakalpojums. Tas nodrošina serveru darbības pārskatu, ģenerējot atskaites par aizņemtās atmiņas daudzumu, procesoru noslodzi, atbildes laikiem un citiem darbības rādītājiem. Šis pakalpojums tiek izmantots bezmaksas versijā, tāpēc dziļākas un detalizētākas analīzes rīki nav pieejami, kā arī statistika ir pieejama tikai par pēdējām 24 darbības stundām.

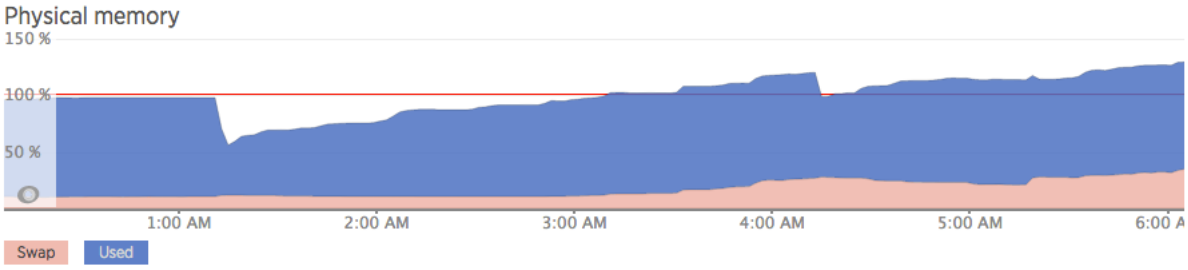
3.2. Sākotnējā monitoringa datu analīze

Izmantojot New Relic monitoringa rīku, tika uzņemti lietotnes servera aizņemtās atmiņas rādījumi. Apskatot 24 stundu periodu darba dienā, sākot no plkst. 14.00 (3.1. att.), redzams, ka nakts laikā servera atmiņas aizpildījums ir ap 70% (grafika laiki norādīti citā laika zonā).



3.1. att. **Lietotnes atmiņas aizpildījums 24 stundu laikā**

Savukārt darba laikā – abos grafika galos – atmiņas aizpildījums sasniedz un pat pārsniedz 100 procentu robežu. 100 procentu robežas pārsniegšana nozīmē, ka serveris sāk izmantot pārneses datni uz cietā diska. Lietotne it kā turpina darbību, bet būtībā tās lietošana ir neiespējama, jo darbības ar pārneses datni aizņem pārāk daudz laika.



3.2. att. **Lietotnes atmiņas aizpildījums darba dienas vidū**

Apskatot citas darba dienas laiku no plkst. 10.00 līdz 16.00 (3.2. att.), grafika vidū redzams lineārs aizņemtās atmiņas pieaugums.

Atmiņa laikā tiek aizpildīta lineāri, līdz pārsniedz 100% aizpildījuma robežu. Pēc tam grafikā redzams neliels samazinājums, kas liecina par drazu savācēja darbību. Tomēr atbrīvota tiek tikai neliela daļa atmiņas, kas nav pietiekami normālai lietotnes darbībai. Grafikā redzams arī pārneses datnes lielums. Tā kļūst lielāka brīdī, kad atmiņas daudzums sasniedz 100%. Lineārais atmiņas aizpildījuma pieaugums laikā, kad lietotni izmanto visintensīvāk, kalpo par pamatu aizdomām par atmiņas noplūdi.

Apskatot atsevišķi tīmekļa servera un fona darbu procesa atmiņas patēriņu, bija redzams, ka fona darbu process spēj patērēt līdz vienam gigabaitam atmiņas, bet drazu savācējs atmiņu pēc tam atbrīvoja. Savukārt Puma tīmekļa servera instances katra atsevišķi spēja aizņemt pat 1,5 gigabaitus atmiņas. Tika nolemts sākotnēji koncentrēties uz tīmekļa servera darbību.

3.3. Slodzes testi ar GET pieprasījumiem

Lai varētu veikt testus atmiņas noplūžu meklēšanai, tika pieņemts lēmums šos testus veikt uz atsevišķas darbstacijas. Ruby lietotnes izmanto moduļu menedžeri, lai nodrošinātu to, ka lietotnes gan uz izstrādes laikā, gan produkcijas vidē izmanto vienu un to pašu versiju moduļus, kā arī pašu Ruby versiju. Tāpēc atdarinātā produkcijas vide uz testa darbstacijas pēc šiem parametriem bija vienāda ar īsto produkcijas vidi. Tika atspējoti pāris moduļi, kas bija atbildīgi par kļūdu ziņošanu un citām monitoringa aktivitātēm, jo tie produkcijas vidē sūta datus uz vairākiem monitoringa trešās puses servisiem.

Lai vēl vairāk pietuvinātu līdzību ar produkcijas vidi, šai testa videi bija pieejama produkcijas vides datu bāzes kopija. Tātad testi tika veikti ar produkcijas videi identiskiem datiem.

3.1. tabula

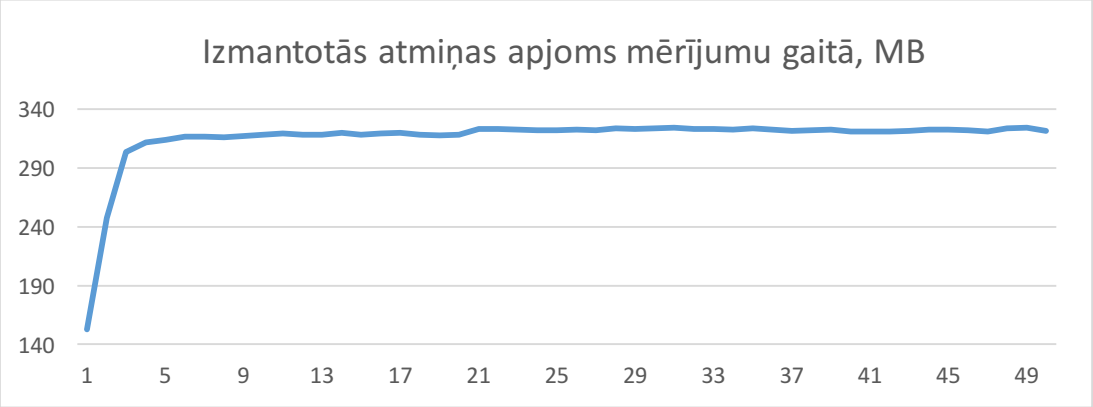
Produkcijas un testa vides parametru salīdzinājums

Parametrs	**Produkcijas vide**	**Testa vide**
Operētājsistēma	Ubuntu 14.04	macOS Sierra 10.12.3
RAM	3,75 GB	10 GB
Procesors	Intel Xeon E5-2666 v3, 2 vCPU	2,7 GHz Intel Core i7
Ruby versija	2.2.3	2.2.3
Tīmekļa servera Puma versija	3.8.2	3.8.2

Kā redzams 3.1. tabulā, galvenās atšķirības starp testa un produkcijas vidēm ir operētājsistēma, operatīvā atmiņa un procesors. Ruby valodas versija un tīmekļa servera Puma versijas ir vienādas.

Lai gūtu priekšstatu par atmiņas patēriņa testēšanu, sākotnēji autors izmantoja "Derailed benchmarks" Ruby moduli. Kā minēts iepriekš, šis modulis piedāvā dažādas atmiņas lietojuma mērīšanas metodes, bet šajā gadījumā svarīgā iespēja bija produkcijas vides simulācija un vienādu, secīgu GET pieprasījumu veikšana lietotnei. Ik pēc 5 sekundēm šis modulis darbības laikā veic lietotnes aizņemtās atmiņas mērījumu. Tā kā pieprasījumi ir vienādi, un to ir daudz, tad tiem vajadzētu uzskatāmi parādīt atmiņas noplūdi, ja lietotnes darbībā tāda būtu.

Testējamajā lietotnē daļa funkcionalitātes ir līdzīga viena otrai. Piemēram, pasūtījumu, klientu un citu objektu saraksta parādīšanai ir izmantots viens un tas pats modulis un darbības principi. Tā kā šī skata veids ir viens no visvairāk izmantotajiem lietotnē, tad pirmie mērījumi tika veikti pasūtījumu saraksta pirmās lappuses attēlošanai. Veicot testu, tika iegūti Ruby aizņemtās atmiņas mērījumi ik pa piecām sekundēm. Pēc 500 pieprasījumiem darbība tika pārtraukta. Iegūtajā grafikā (3.3. att.) redzams atmiņas aizpildījums testu laikā.



3.3. att. **Pasūtījumu saraksta slodzes testa rezultāti**

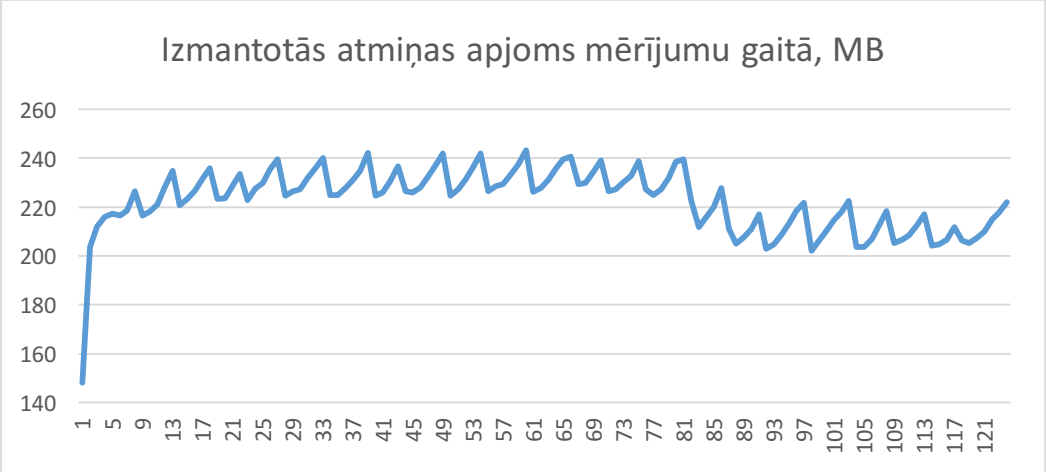
Sākotnējais straujais atmiņas izmantojuma pieaugums skaidrojams ar datu fragmentāciju, bet pēc piektās mērījumu reizes (tātad pēc 25 sekundēm) atmiņas izmantojums vairs praktiski nemainās. Tas nozīmē, ka atmiņas noplūdes šajā lietotnes funkcionalitātē nav, un atmiņa, kas nepieciešama tieši šī saraksta attēlošanai, ir niecīga, jo nav novērojama liela drazu savācēja darbība, kas izpaustos kā atmiņas samazinājums.

Jāpiebilst, ka ar šo pašu "Derailed benchmarks" moduli tika izmērīts atmiņas daudzums, kas vajadzīgs tikko startētai lietotnei kopā ar visiem moduļiem. Un tas bija 133 megabaiti. Tātad, pēc lietotnes darbības sākšanas tai ir vajadzīgi vēl aptuveni 170 megabaiti, lai varētu attēlot pasūtījumu sarakstu.

Tā kā lielākā daļa darbību lietotnē ir saistītas ar pasūtījumiem, un tās pēc pieredzes strādājot ar šo lietotni ir vislaikietilpīgākās, tad, ņemot vērā Ričarda Šnēmana ieteikumu sākt tieši ar laikietilpīgāko funkcionalitāti, tika pieņemts lēmums veikt šāda veida testus ar dažāda veida pieprasījumiem saistībā ar pasūtījumiem.

Kā visbiežāk apmeklētais skats tika izraudzīts pasūtījuma informācijas skats. Tajā

redzama gandrīz visa lietotājam neieciešamā informācija par pasūtījumu, izmaksām un pasūtītajām precēm. Veicot mērījumus ar sīki aizpildītu pasūtījumu, kam ir pievienots daudz preču, tika iegūts grafiks (3.4 att.), kurā skaidri redzama drazu savācēja darbība. Grafikā redzamā līnija ir ar izteiktiem kāpumiem un kritumiem, kas, kā noprotams, ir strauja atmiņas aizņemšana un tūlītēja atmiņas atbrīvošana. Tas nozīmē, ka šī skata attēlošanai vajadzīgi ievērojami atmiņas resursi. Pēc vairākiem simtiem pieprasījumu redzams, ka atmiņas izmantojumam nav tendence pieaugt. Grūti izskaidrojams ir redzamais atmiņas izmantojuma samazinājums pēc 81. mērījuma, bet tam lielāka uzmanība netika pievērsta, jo pastāvīgs atmiņas izmantojuma pieaugums netika konstatēts.

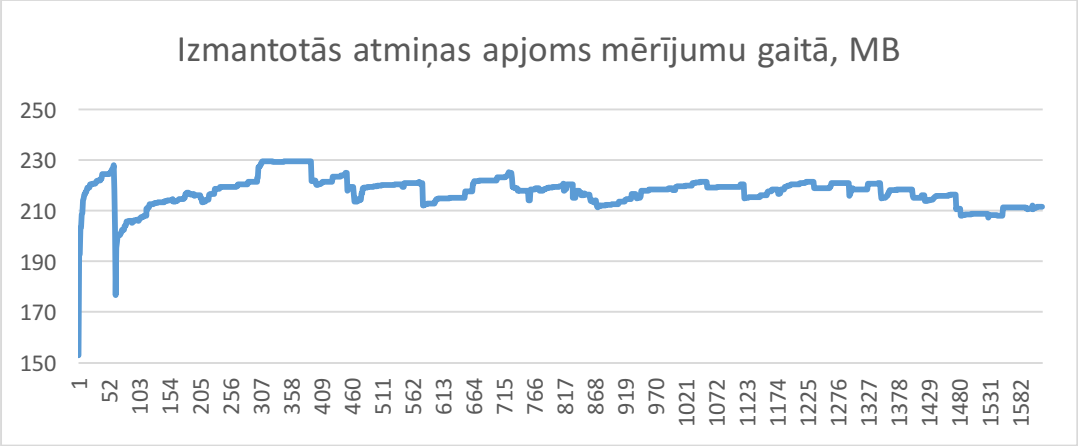


3.4. att. **Pasūtījuma detalizētā skata slodzes testa rezultāti**

Līdzīgi bija arī ar citiem statiskiem mērījumiem. Lai arī testos šie izsaukumi bija daudz vairāk nekā dienā vidēji lietotne saņem viena veida izsaukumu, un tam vajadzēja izteikti parādīt atmiņas noplūdi, testi šīs noplūdes nejauši izvēlētiem skatiem neparādīja. Bija skaidrs, ka ar nejauši izvēlētiem skatiem testēšana nebūs produktīva. Skatu ir daudz, atmiņas noplūdes, ja tādas ir, var būt jebkurā no tiem.

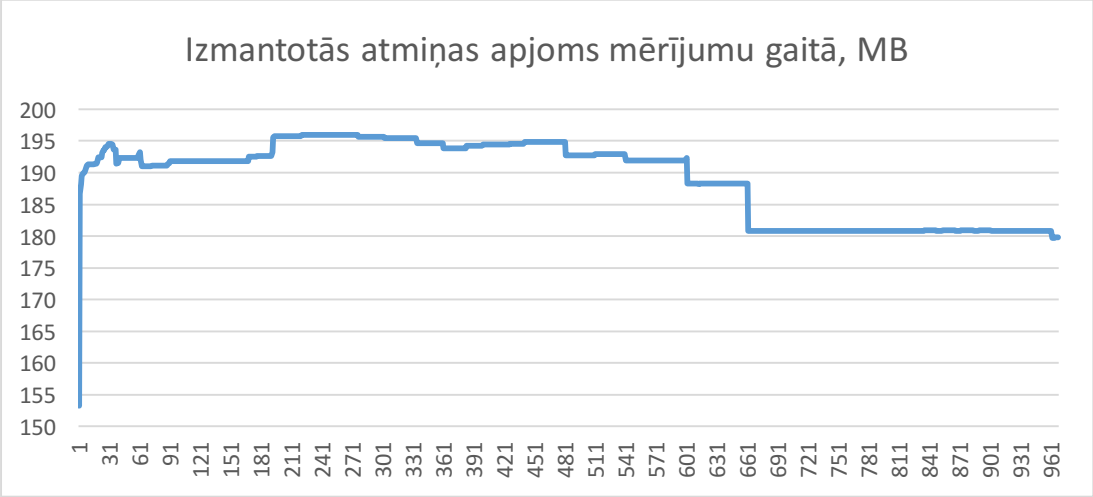
Tomēr, mēģinot apstiprināt aizdomas par kādu datņu eksporta moduļiem, tika pieņemts lēmums notestēt katru no tām, veicot eksporta pieprasījumus lietotnei, lai saņemtu datnes dažādos formātos.

Kā pirmais tika izvēlēts xls formāta eksportētājs. Lai to notestētu, autors veica slodzes testus ražošanas plānotāja dienas plāna izdrukai. Sākotnējā testā grafiks nebija izlīdzinājies, tāpēc tika veikti mērījumi ar daudz lielāku – 10 000 – mērījumu skaitu. Jaunajā grafikā (3.5. att.) redzams, ka izmantotās atmiņas apjoms nepieaug. Lai arī tas ir salīdzinoši mainīgs, aizdomas par atmiņas noplūdēm xls eksportētāja modulī neapstiprinājās.



3.5. att. **xls eksporta slodzes testa rezultāti**

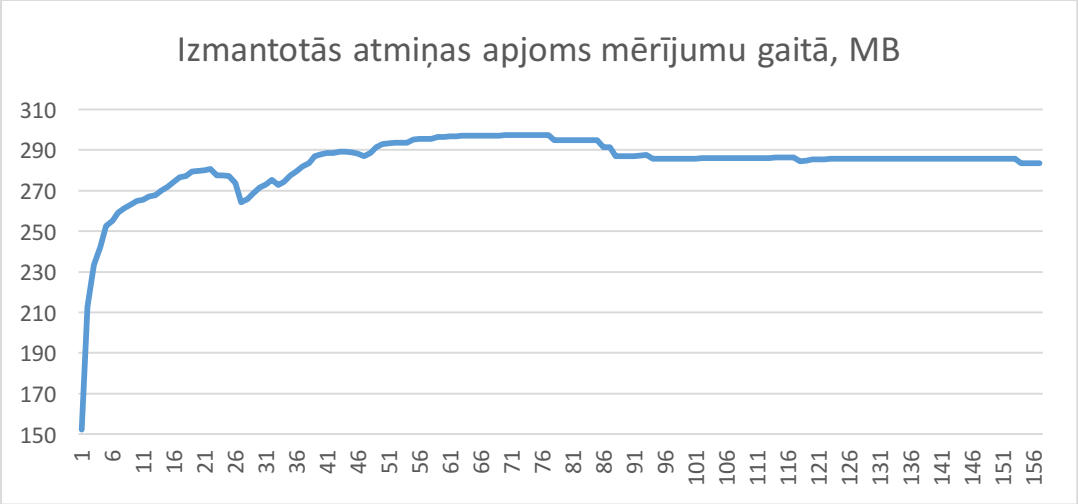
Līdzīgi testi tika veikti ar pdf formāta eksportētāju. Veicot slodzes testu pasūtījuma informācijas izdrukai pdf formātā, tika iegūts grafiks (3.6. att.).



3.6. att. **pdf eksporta slodzes testa rezultāti**

Grafikā redzama ievērojami lielāka atmiņas aizpilde sākumā, salīdzinot ar grafika beigu daļu, kurā atmiņas aizpilde izlīdzinās un kļūst konstanta. Iespējams, tas saistīts ar datu fragmentācijas samazināšanos. Tomēr skaidri redzams, ka atmiņa laika gaitā netiek aizņemta arvien vairāk un vairāk. Turklāt jāņem vērā, ka šāda veida izdrukas pieprasījumu skaits dienas laikā var sasniegt pāris simtus, bet noteikti ne tik daudz, kā tas tika veikts šajā slodzes testā.

Kā pēdējais datņu eksporta tests bija ar docx datņu ģenerēšanu. Docx datnes lietotnē lieto, lai varētu veidot dažādas veidlapas ar vietturiem, kurus lietotne pēc tam var aizstāt ar pasūtījuma vai preču vajadzīgo informāciju dažādai dokumentācijai. Iegūtais grafiks (3.7. att.) raksturā atšķīrās no iepriekš iegūtajiem, tomēr to kopīgā iezīme ir sākotnējā atmiņas aizpildīšanās šķietami vairāk, nekā nepieciešams, jo pēc tam atmiņas aizpildījums samazinās un kļūst vienmērīgs.

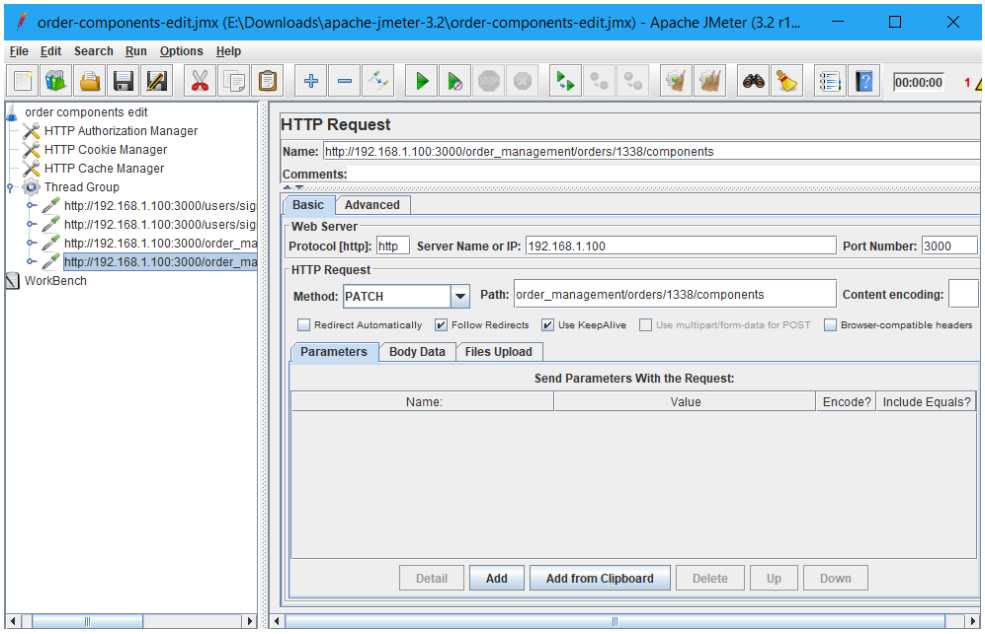


3.7. att. **docx eksporta slodzes testa rezultāti**

Arī docx datņu eksports neuzrādīja nekādas atmiņas noplūžu pazīmes.

3.4. Slodzes testi ar POST pieprasījumiem

Tīmekļa lietotnes atbild ne tikai uz GET pieprasījumiem, bet pieņem arī dažāda veida datus apstrādei un saglabāšanai ar POST un PATCH pieprasījumiem. Tā kā sākotnējie testi ar GET pieprasījumiem neliecināja par atmiņas noplūdēm, bija vajadzība notestēt arī šos datu ielādes peiprasījumus. "Derailed benchmarks" modulis piedāvā tikai GET pieprasījumu testus, tāpēc bija jāmeklē citi veidi, kā automatizēti veikt lielu daudzumu pieprasījumu.



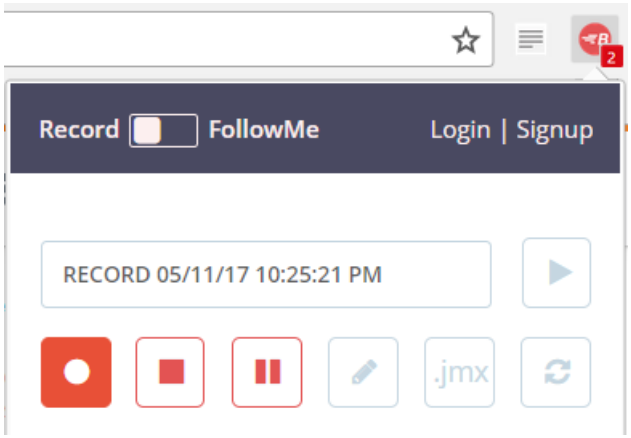
3.8. att. **JMeter lietotnes grafiskā lietotāja saskarne**

Lai to varētu izdarīt, tika izmantota Apache programmatūras fonda izstrādātā JMeter lietotne (3.8. att.).

Šajā lietotnē var sastādīt testa plānu ar pieprasījumiem. Katram pieprasījumam var norādīt tipu, adresi, kā arī nosūtāmos datus. Bija svarīgi, lai ar šo lietotni varētu realizēt sesijas uzturēšanu un lietotāja autentifikāciju. JMeter automātiski piedāvā sīkdatņu uzturēšanu, tāpēc tas nebija jākonfigurē.

Ruby on Rails lietotnes formās automātiski ievieto pazīšanas zīmi slēptā formas laukā. Ja ienāk pieprasījums ar formas datiem, bet tajos nav iepriekš nosūtītā pazīšanas zīme, lietotne uzskata, ka notiek mēģinājums viltot pieprasījumu un to neapstrādā. JMeter lietotne dod iespēju atrast specifisku tekstu tīmekļa lietotnes atbildē un saglabāt to nākamajiem pieprasījumiem. Ar šīs iespējas palīdzību var veikt pieprasījumu pēc pieteikšanās formas, saglabāt pazīšanas zīmi un nosūtīt pieteikšanās pieprasījumu, iekļaujot šo pazīšanas zīmi kopā ar lietotājvārdu un paroli. Turklāt pieteikumus var grupēt un norādīt to izpildīšanas reižu skaitu. Tā kā pieteikšanos vajag veikt tikai vienu reizi, šos divus pieprasījumus sagrupēja, ievietoja apstrādājamās rindas sākumā un norādīja, ka tā jāizpilda tikai vienreiz.

Lai būtu vieglāk izveidot testēšanas plānu, tā pieprasījumiem un to saturu, tika izmantota mākoņskaitļošanas testēšanas pakalpojuma BlazeMeter izveidotā pārlūka pievienojumprogramma, kas saglabā visus pārlūkprogrammā veiktos HTTP pieprasījumus ar datiem un ļauj tos eksportēt JMeter lietotnei saprotamā formātā [16]. Tās lietotāja grafiskā saskarne satur intuitīvas ieraksta sākšanas un apturēšanas pogas (3.9. att.).



3.9. att. **BlazeMeter pievienojumprogrammas lietotāja grafiskā saskarne**

Atverot eksportēto datni ar JMeter, reģistrētos pieprasījumus var labot, dzēst un kārtot vajadzīgajā secībā. Šī pievienojumprogramma ļāva ietaupīt laiku testa plānu sastādīšanā.

Lai uzņemtu lietotnes atmiņas patēriņu laikā, tika izmantots Ričarda Šnēmana izveidotais modulis "GetProcessMem", ko izmanto arī "Derailed benchmarks" modulis. Tas ir mazs rīks, kas iegūst servera procesa atmiņas patēriņu attiecīgajā laika brīdī. Tika izveidota papildu funkcija (3.10. att.), kuras uzdevums bija fiksēt atmiņas patēriņu un saglabāt to atsevišķā datnē.

Šo funkciju izsauca lietotnes kontrolieris pēc katras pieprasījuma apstrādes.

```
def print_mem
  mem = GetProcessMem.new
  logger = Logger.new("#{Rails.root}/log/my.log")
  logger.info mem.mb
end
```

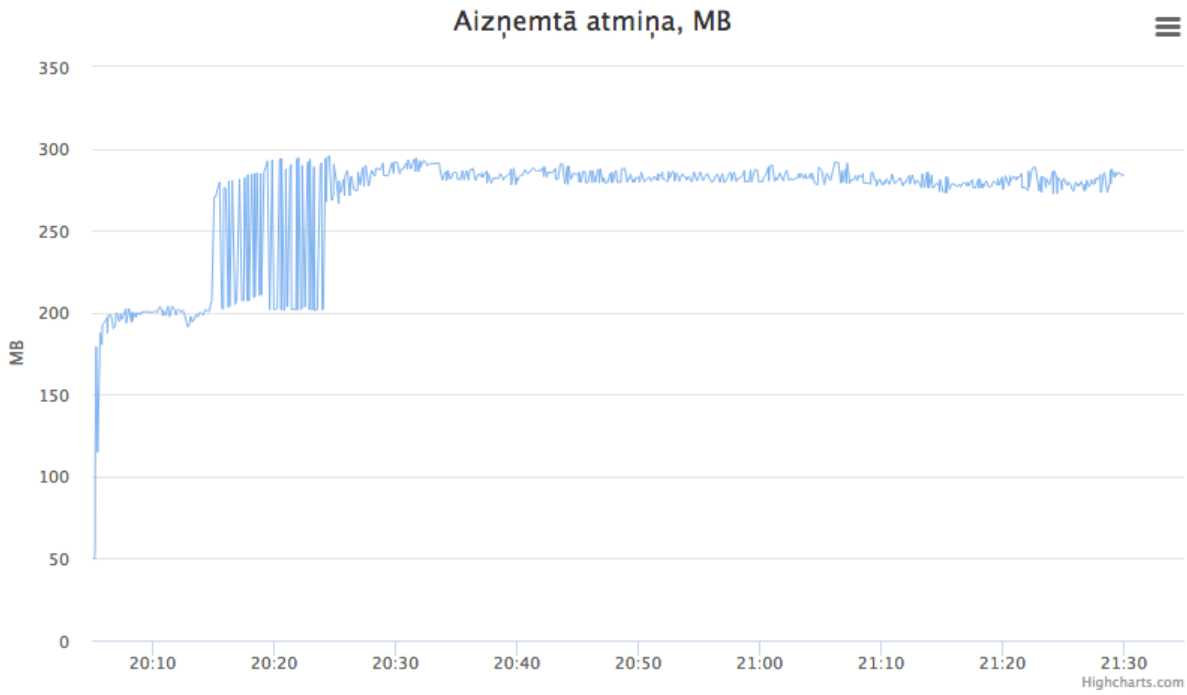
3.10.att. **Funkcija atmiņas patēriņa izmaiņu fiksēšanai**

Rezultātā tika iegūta žurnāla datne ar klasisku Ruby žurnāla uzbūvi (3.11. att.).

```
I, [2017-05-12T19:15:11.583464 #38289]  INFO -- : 282.609375
I, [2017-05-12T19:15:12.928967 #38289]  INFO -- : 288.625
I, [2017-05-12T19:15:14.122336 #38288]  INFO -- : 273.80078125
```

3.11. att. **Iegūto atmiņas mērījumu rezultāta fragments**

Ņemot vērā to, ka pasūtījumu apstrāde ir visilgākā un resursus prasīgākā, testēšana ar formu nosūtīšanu tika sākta tieši ar pasūtījumiem. Produkcijas vidē lietotne saņem ap 100 šādiem pieprasījumiem, bet testu laikā formas ielādes un nosūtīšanas pāri tika veikti 500 reižu. Pirmais tests tika veikts ar pasūtījuma preču formas apstrādi.



3.12. att. **Atmiņas patēriņš testu laikā**

Grafikā (3.12. att.) redzams, ka sākotnēji patērētā atmiņa svārstās ap 200 megabaitiem. Nākamajā posmā atmiņa tiek pieprasīta un uzreiz atbrīvota. Šīs svārstības ir ļoti straujas, papildu pieprasītās un atbrīvotās atmiņas apjoms ir gandrīz 100 megabaiti. Tas ir pretrunā ar Scout servisa emuārā rakstīto, ka Ruby atmiņu atbrīvo nelabprāt. Tomēr pēc tam pieprasītais atmiņas daudzums turpina svārstīties ap 280 megabaitiem. Tika izdarīts secinājums, ka atmiņas noplūde šī tipa funkcionalitātē nav, kā arī datu apstrāde prasa aptuveni tikpat atmiņas, cik iepriekš veiktajos testos ar GET tipa pieprasījumiem.

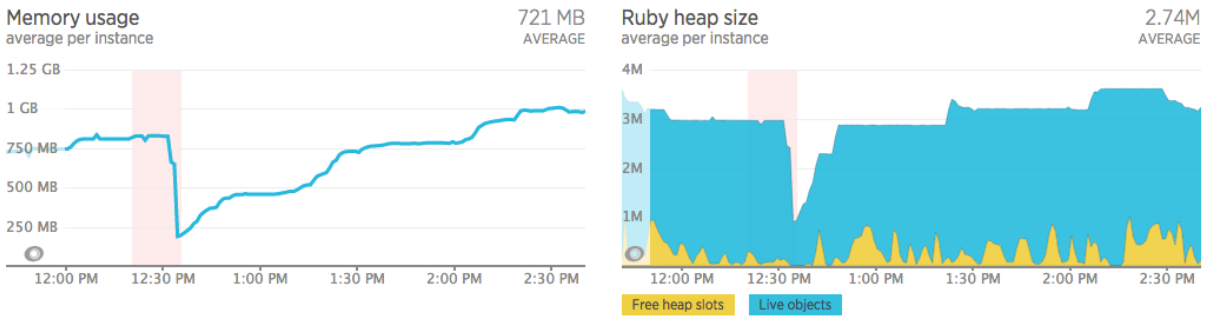
Līdzīgi tika testētas pāris citas formu apstrādes, un rezultāti neliecināja par jebkādām atmiņas noplūdēm.

Neskatoties uz BlazeMeter pievienojumprogrammas atvieglojumu JMeter testa plānu sagatavošanā, tas tomēr bija laikietilpīgs un sarežģīts process, tāpēc tika notestēti tikai daži šādi pieprasījumi. Sāka kļūt skaidrs, ka pa vienam testējot atsevišķus pieprasījumus, kas aizņem zināmu laiku, atmiņas noplūžu meklēšana nav efektīva.

3.5. Produkcijas vides rādītāju analīze

Atsevišķas funkcionalitātes, kas izraudzīta tikai uz aizdomu pamata, testēšana nenesa nekādus rezultātus. Bija vajadzība izvēlēties citu atmiņas noplūžu meklēšanas stratēģiju. Tika pieņemts lēmums meklēt korelāciju starp produkcijas vides pieprasītās atmiņas daudzuma izmaiņām un to, kādi attiecīgajā laika posmā ir bijuši pieprasījumi serverim.

Pēdējo 24 stundu atmiņas lietojuma, kā arī Ruby objektu kaudzes izmēru un drazu savācēja darbības var iegūt, izmantojot New Relic tiešsaistes servisu (3.13. att.). Attēlā redzamie grafiki rāda atmiņas lietojumu un Ruby objektu kaudzes izmēru laika gaitā vienai servera instancei. Aplūkojot grafiku redzam, ka kopā augstākajā punktā abas servera HTTP pieprasījumu instances patērē 2 gigabaitus atmiņas.



3.13. att. **New Relic servera darbības rādītāji**

Tomēr šajā rīkā attēlotie grafiki ir mazi, to izmantošana korelācijas saskatīšanai nebija uzticama. Grafiki tiek zīmēti ar JavaScript palīdzību, izmantojot dinamiski ielādētus datus JSON formātā. Izmantojot pārlūkprogrammas izstrādāja rīkus, šos datus var lejupielādēt un izmantot ērtākā veidā. Dati satur laika un izmantotās atmiņas vērtību pārus.

Ruby on Rails lietotnes uztur darbības žurnālu datni, kurā, lietotne veic veikto darbību pierakstu. Starp šiem ierakstiem, ja nav veiktas konfigurācijas izmaiņas, atrodas arī visu HTTP pieprasījumu atzīmes ar laikiem. Attālināti pieslēdzoties lietotnes produkcijas vides serverim,

tika iegūts šo pieprasījumu saraksts (3.14. att.).

```
[36m2017-05-16T20:56:50.089182056Z app[web.1]:[0m Started GET "/" at 2017-05-16 20:56:50 +0000
[36m2017-05-16T20:56:52.719465146Z app[web.1]:[0m Started GET "/order_management" at 2017-05-16 20:56:52 +0000
[36m2017-05-16T20:56:56.494379246Z app[web.1]:[0m Started GET "/users/sign_in" at 2017-05-16 20:56:56 +0000
[36m2017-05-16T20:56:56.573047933Z app[web.1]:[0m Started GET "/" at 2017-05-16 20:56:56 +0000
```

3.14. att. **Iegūtais pieprasījumu saraksts no servera žurnāla**

Lai varētu noskaidrot, kuri pieprasījumi ir notikuši visbiežāk, tika izmantoti komandrindas teksta meklēšanas, aizstāšanas un kārtošanas rīki (3.15. att.).

```
grep -oE '(GET|POST|PATCH|DELETE) ".*"' local.log | sed 's/"//g' | sed 's/\?.*//' \
| sed -E 's,/[0-9]+,/X,' | sort | uniq -c | sort -r -n
```

3.15. att. **Komandu ķēde pieprasījumu grupēšanai un kārtošanai**

Rezultātā autors ieguva pēc biežuma sakārtotu pieprasījumu sarakstu. Lai varētu apvienot viena tipa pieprasījumus, kuros ir objekta identifikators, ar "sed" funkcijas palīdzību šie identifikatori tika aizstāti ar simbolu "X" (3.16. att).

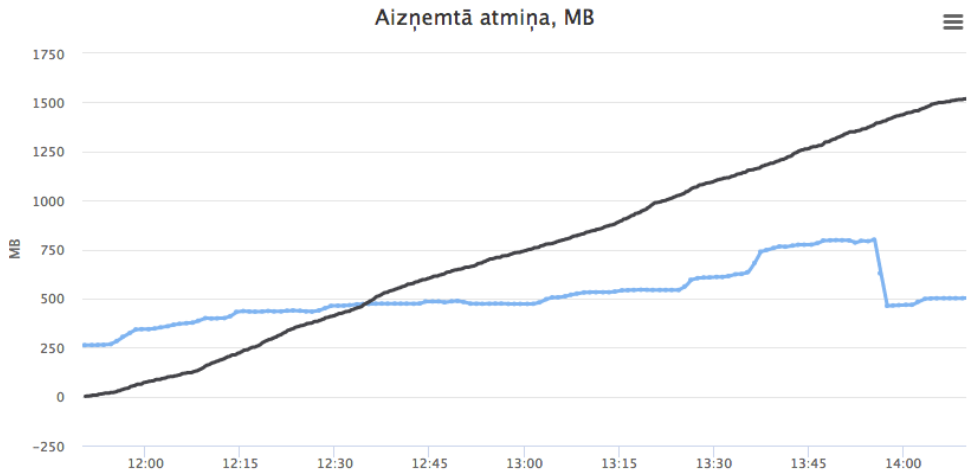
```
1623 GET /users/sign_in
 845 GET /
 313 GET /order_editor/order_item_configuration.json
 226 GET /order_management/contacts/autocomplete
 182 GET /order_management/orders/X
 170 GET /production_planner/production_operations.json
  58 GET /product_management/product_packs/autocomplete_products
  53 GET /order_management/orders
```

3.16. att. **Pēc skaita sakārtoti pieprasījumi**

Visbiežākie pieprasījumi bija lietotāja autentifikācijas formai un sākumlapai. Tas skaidrojams ar to, ka lietotnes darbības uzraudzībai New Relic serviss veic šos pieprasījumus, lai pārbaudītu, ka lietotne darbojas un spēj apstrādāt pieprasījumus.

Lai varētu saskatīt korelāciju starp pieprasījumiem un atmiņas pieaugumu, kas liecinātu par atmiņas noplūdi, no iegūto pieprasījumu žurnāla tika atlasīti viena veida pieprasījumi ar to laikiem. Tad šie laiki tika salikti vienā grafikā ar iepriekš iegūto atmiņas lietojuma grafiku. Katrs piefiksētais laiks palielina grafika līkni par vienu iedaļu. Iegūtajam grafikam būtu skaidri jāparāda korelāciju starp pieprasījumiem un atmiņas izmantojuma palielināšanos. Ja pieprasījums izraisa atmiņas noplūdi, tad atmiņas lietojums straujāk palielināsies brīžos, kad šie pieprasījumi tiks izsaukti biežāk.

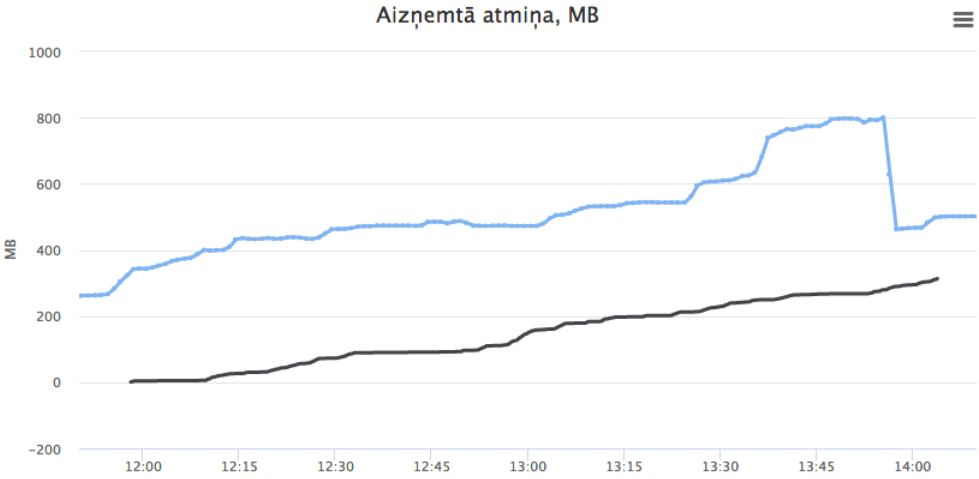
Kā pirmais no pieprasījumu veidiem tika testēta lietotāja autentificēšanās forma. Rezultāti apskatāmi 3.17. attēlā. Tumšā līkne norāda uz pieprasījumu skaita izmaiņām, bet gaišā ilustrē izmantotās atmiņas apjomu.



3.17. att. **Izmantotās atmiņas pieaugums un autentificēšanās formas pieprasījumi**

Grafikā redzams, ka New Relic serviss veic pieprasījumus regulāri, jo grafika līkne tam ir gandrīz taisna. Divu stundu laikā tas veic gandrīz 1500 pieprasījumus. Grafikā nav redzama sakarība starp šiem pieprasījumiem un atmiņas izmantojuma palielināšanos. Tas ir saprotami, jo lietotne, atbildot uz šiem pieprasījumiem, ģenerē tikai autentifikācijas formu un datus neapstrādā.

Nākamais biežākais pieprasījuma veids uzņemtajā laika intervālā ir "/order_editor/order_item_configuration.json". To izmanto, lai pēc preces konfigurācijas aprēķinātu detaļu izmērus, darāmos darbus un izmaksas. Šie pieprasījumi notiek dinamiski, lietotājam izmainot kādu preces parametru formā. Apskatītajā divu stundu laika intervālā šāds pieprasījums lietotnei tikai veikts 313 reižu.



3.18. att. **Izmantotās atmiņas pieaugums un preču konfigurācijas pieprasījumi**

Grafikā redzams, ka šāda veida pieprasījumi izdarīti diezgan vienmērīgi, atmiņas

palielināšanos ar tiem nevar sasaistīt.

Šādā veidā grafiski attēlojot pieprasījumu biežumu, tika pārbaudīti visi visvairāk izsauktie pieprasījumi. Tomēr saskatīt jebkāda veida saistību ar atmiņas palielināšanos neizdevās. Pieprasījumi notiek pamīšus, ir iespējams, ka lielo atmiņas pieaugumu veido dažādi resursus prasoši izsaukumi reizē. Bija iespējams, ka grafikā redzamais straujais atmiņas kāpums nav saistīts ar vairākiem viena veida izsaukumiem. Tikpat labi tas var būt atmiņas burbulis, ko veido daži vai tikai viens īpašs izsaukums.

Bija skaidrs, ka arī ar šādu metodi noteikt iespējamu atmiņas noplūdi kādā funkcionalitātē nav iespējams.

3.6. Pieprasījumu reproducēšana

Iepriekšējās pētīšanas metodes nebija nesušas gaidītos rezultātus, tādēļ tika pieņemts lēmums reproducēt pieprasījumus, izmantojot to pašu servera žurnālu. Bija cerība, ka tādā veidā varētu gūt lielāku izpratni par atmiņas patēriņu. Lai to izdarītu, žurnāla ieraksti tika apstrādāti tā, lai katra teksta rindiņa saturētu tikai pieprasījuma URL vietrādi.

Pieprasījumu secīgai un automatizētai izpildei bija jāuzraksta Ruby skripts. Tā kā aplūkojamā lietotne darbības izpilda tikai autentificētam lietotājam, bija vajadzīgs veids, kā veikt autentifikāciju un uzturēt sesiju. Tam noderēja Ruby modulis “Mechanize”.

```
BASE = 'https://example.com'

def url(s)
  "#{BASE}#{s}"
end

agent = Mechanize.new
page = agent.get(url '/users/sign_in')
form = page.forms.first
form['user[email]'] = 'username'
form['user[password]'] = 'password'
page = agent.submit(form)

100.times do
  counter = 0
  IO.foreach("test_plan.txt") do |line|
    begin
      page = agent.get(url line)
      puts counter += 1
    rescue
      puts "FAIL: #{line}"
    end
  end
end
```

3.19. att. **Skripts automatizētai pieprasījumu veikšanai ar lietotāja autentificēšanu**

Tas vienkāršo ne tikai sesijas uzturēšanu, bet ar tā palīdzību var aizpildīt formu laukus,

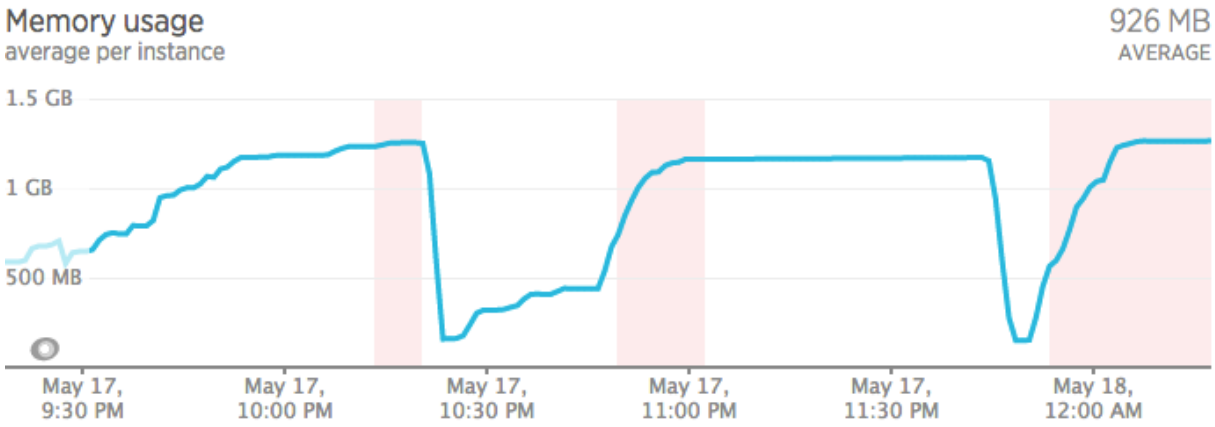
kas noderēja lietotāja autentifikācijas procesā (3.19. att.). Kā redzams attēlā, ar "Mechanize" moduļa palīdzību veikt lietotāja autentificēšanu ir ļoti vienkārši. Modulis atpazīst saņemto HTML iezīmēšanas valodu, un ļauj aizpildīt un nosūtīt tajā esošo formu.

Tālāk skripts ciklā ielasa pa viena rindiņai no datnes, kurā ir atrodas tikai URL vietrādes, kas iepriekš iegūtas no servera žurnāla. Veiksmīga pieprasījuma gadījumā skripts komandrindā izdrukā attiecīgā URL vietrāža rindiņas, kurā tā atrodas, numuru. Neveiksmīga pieprasījuma gadījumā URL vietrādis tiek izvadīts uz ekrāna.

Lai lieki neietekmētu testu veikšanu, bija vajadzīga otra darbstacija, no kuras veikt pieprasījumus lietotnei. Tā kā tāda nebija pieejama, pieprasījumi tikai veikti lietotnei produkcijas vidē. Bija svarīgi netraucēt lietotnes darbu darba laikā, lai lietotāji nejustu nekādas darbības problēmas, tāpēc testi tika veikti nakts laikā.

Vēlreiz aplūkojot 3.18. attēlu, var redzēt strauju atmiņas kāpumu laikā starp plkst. 13:30 un 13:45. Pieprasītais atmiņas apjoms aug par aptuveni 200 megabaitiem. Tā kā tas ir straujākais kāpums, tas bija labs sākuma punkts testu veikšanai. No žurnāla tika atlasīti pieprasījumi attiecīgajā laika sprīdī, no kuriem tika sagatavota testa plāna datne. Ar sagatavotā skripta palīdzību testa plānā esošie pieprasījumi tika veikti vairākus desmitus reižu. Apskatot New Relic atmiņas aizpildījuma monitoringa grafiku, bija redzams, ka atmiņas lietojums tiešām ir liels, strauji kāpj un neizlīdzinās. Bija pamats aizdomām, ka kāds no testa plānā esošajiem pieprasījumiem izraisa atmiņas noplūdi.

No testa plāna bija jāatlasa viena veida pieprasījumi un jāatkārto tests tikai ar šiem pieprasījumiem. Atklājās, ka vainīgā funkcionalitāte ir pasūtījumu saraksta atveidošana. 3.20. attēlā redzams New Relic servisa ģenerētais lietotnes atmiņas lietojuma grafiks.



3.20. att. **New Relic servisa sagatavotais atmiņas lietojuma grafiks**

Laikā starp plkst. 9:30 un 10:30 tika veikts sākotnējais tests ar visiem pieprasījumiem no servera žurnāla aizdomīgajā laika intervālā. Tā kā vainīgie pieprasījumi mijās ar citiem, tad aizpildītās atmiņas daudzums nav tik straujš, bet tāpat dažu desmitu minūšu laikā pārsniedza 1 gigabaita atzīmi. Mirkli vēlāk New Relic sāka ziņot par neapmierinošu servera atbildes laiku.

Laika intervāli, kad serveris nespēj pienācīgi ātri atbildēt, ir iezīmēti grafikā.

Pēc pirmajiem testiem serveris bija manuāli jārestartē, tāpēc grafikā ap plkst. 10:20 redzams straujš izmantotās atmiņas samazinājums. Tā kā pasūtījuma saraksta pieprasījumi bija visuzkrītošākie servera žurnālā, tad tie bija pirmie, kurus izraudzīja atsevišķai testēšanai. Lai netērētu laiku aizdomu neapstiprināšanās gadījumā, pirmie atsevišķie testi bija ar nelielu daudzumu pieprasījumu. Tas redzams grafikā laikā starp plkst. 10:20 un 10:45, kad līkne kāpj vairākus nelielus soļus uz augšu. Izskatījās, ka šie pieprasījumi varētu būt pie vainas, tāpēc ap plkst. 10:45 skripts izpildīja lielāku daudzumu šo te pieprasījumu, kā rezultātā patērētā atmiņa strauji auga. Bija skaidrs, ka ar šiem pieprasījumiem dažu minūšu laikā var panākt to, ka serveris ir spiests izmantot pārneses datni un līdz ar to panākt faktisku servera darbības apturēšanu.

Aptuveni plkst. 11:45 tika veikts vēl viens servera restarts, un vēl viens to pašu pieprasījumu tests, lai vēlreiz pārliecinātos par šo pieprasījumu slikto ietekmi uz servera darbību, ar atšķirību, ka tika veikti uzreiz 500 pieprasījumi viens aiz otra, bez pauzēm pa vidu. Tāpēc grafika līkne ir stāvāka nekā iepriekšējo reizi.

Apskatot tuvāk šos pieprasījumus, atklājās, ka tie satur daudzus GET parametrus (3.21 att.).

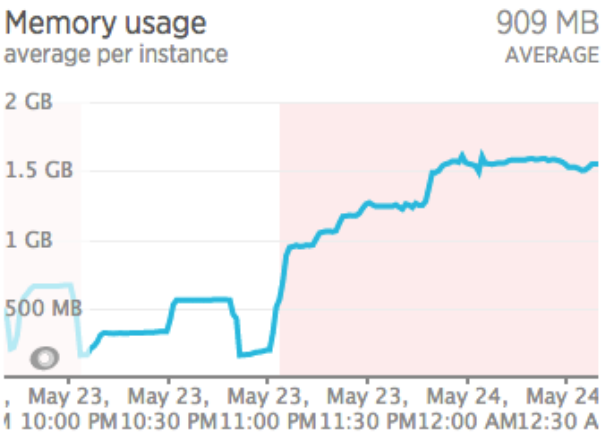
```
/order_management/orders?utf8=✓
om_og[column_names][]=number
om_og[column_names][]=client_reference
om_og[column_names][]=product_group
om_og[column_names][]=contact
om_og[column_names][]=address_object_name
om_og[column_names][]=address
om_og[column_names][]=responsible_manager
om_og[column_names][]=responsible_for_payer
...
om_og[column_names][]=state
om_og[column_names][]=sale_price_excluding_vat
om_og[column_names][]=paid_amount
om_og[number]=
om_og[responsible_manager_id][]=
state_date_quick_date=
om_og[state_date][]=
om_og[state_date][]=
om_og[contact_id]=
om_og[product_group_id]=4
...
om_og[accounted]=
commit=Atlasīt
```

3.21. att. **Pasūtījumu saraksta pieprasījuma GET parametri**

Pasūtījumu saraksta funkcionalitāte lietotnē ir paplašināta ar meklēšanas un vajadzīgo kolonnu izvēles iespējām. Lietotnes uzstādījumos noteikts, ka saraksta skatā vienlaicīgi var ielādēt līdz 100 objektiem, turklāt katram no šiem objektiem ir piesaistīti daudzi citi objekti, par kuriem lietotāji vēlas zināt, aplūkojot pasūtījumu sarakstu. Kā Scout raksta apskatā minēts,

svarīgi šādos gadījumos novērst N+1 datu bāzes pieprasījumus. Tomēr šajā gadījumā no tiem nevar izvairīties, jo lietotājs pats izvēlas kolonnas, kuras viņš vēlas redzēt. Ja lietotājs izvēlas apskatīt visas iespējamās kolonnas ar saistītajiem pasūtījumiem, katram ielādētajam pasūtījumam var būt nepieciešami pat 5 papildu pieprasījumi datu bāzei. Tātad 500 + 1 pieprasījums saraksta datu iegūšanai. Turklāt jāņem vērā, ka šie ielādēti dati tiek interpretēti Ruby on Rails ietvara noteiktajā kārtībā ar Active Record klasi. Pareizi būtu bijis, ja datu bāzes pieprasījumam dinamiski tiktu pievienoti JOIN nosacījumi attiecīgajām tabulām. Lietotnē datu bāzes pieprasījumā JOIN nosacījumi izveidoti statiski un iekļauj tikai pašas vajadzīgākās saistītās tabulas.

Lai noteiktu, vai šī pasūtījumu saraksta funkcionalitāte tiešām rada atmiņas noplūdes, bija jāizpilda vēl vairāk pieprasījumi. Izpildot divus tūkstošus pieprasījumu pēc kārtas, New Relic atskaites grafikā (3.22. att.) redzams, ka katra no abām servera instancēm ir pieprasījusi ap 1,5 gigabaitu atmiņas, bet šis atmiņas daudzums vairs neaug. Tātad atmiņas noplūde nenotiek, bet pieprasītais atmiņas daudzums ir ļoti liels 100 pasūtījumu tabulas saraksta attēlošanai.



3.22. att. **Atmiņas lietojuma izlīdzināšanās pasūtījumu saraksta testu laikā**

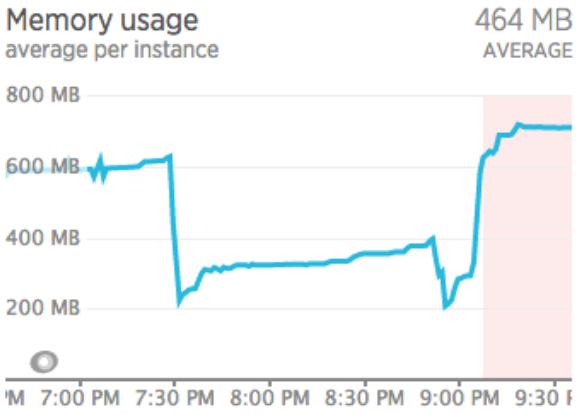
Šie mērījumi krasi kontrastē ar šīs pašas funkcionalitātes testu, kas aprakstīts 3.3. nodaļā, kur "Derailed benchmarks" moduļa testa laikā lietotne nepieprasīja vairāk par 330 megabaitiem. Minētajā testā tika izmantota produkcijas vides datu bāzes kopija, tātad dati bija vienādi. Iespējamās atšķirības var būt skaidrojamas ar dažādajām operētājsistēmām starp produkcijas un test vidi. Bija jāsecina, ka iepriekš veiktie testi nepareizi atspoguļoja lietotnes darbību produkcijas vidē.

Tātad pētāmās lietotnes darbībai ir nepieciešams daudz vairāk atmiņas, nekā sākotnēji uzrādīja testi. Pasūtījumu saraksta skats ir viens no biežāk lietotajiem skatiem, tomēr tas nav vienīgais. Kļuva skaidrs, ka kombinācijā ar citu funkcionalitāti lietotne patērē vairāk atmiņas, nekā serverim ir pieejams.

Apskatot tuvāk Puma tīmekļa servera darbības principus, atklājās, ka katra Puma tīmekļa

servera instance ir spējīga veidot atsevišķus laiksakritīgus pavedienus papildu paralelitātes darbības nodrošināšanai. Puma dokumentācijā teikts, ka jāizvairās atļaut pārāk daudz pavedienu veidošanu, jo tie var izsmelt sistēmas resursus, tai skaitā atmiņu [17]. Pētāmās lietotnes Puma tīmekļa servera uzstādījumos bija noteikts, ka maksimālais pavedienu skaits vienai servera instancei ir 8. Tātad divas servera instances kopā spēj veidot 16 pavedienus.

Lai novērtētu, vai pie lielā atmiņas patēriņa nav vainīgs pārāk liels pavedienu skaits, Puma tīmekļa servera uzstādījumi tika izmainīti, nosakot viena pavediena ierobežojumu katrai instancei. Tika veikti atkārtoti testi ar pasūtījumu saraksta pieprasījumiem. Rezultāti redzami 3.23. attēlā.



3.23. att. **Atmiņas lietojums pēc pavedienu skaita samazināšanas**

Kā redzams, atmiņas patēriņš laika posmā no 9.00 līdz 9.30, kad tika veikti testi, nesasniedza iepriekšējo līmeni. Ja ar 16 pavedieniem kopējais tīmekļa servera atmiņas patēriņš bija vairāk nekā 3 gigabaiti, tad ar 2 pavedieniem tas sasniedza aptuveni 1,5 gigabaitus. Varam izskaitļot, ka samazinājums nav proporcionāls, jo tad patērētajai atmiņai būtu bijis jābūt 8 reizes mazākai nekā iepriekš, bet tā ir aptuveni tikai uz pusi mazāka. Tomēr tas liecina, ka pie lielā atmiņas patēriņa ir vainojams arī pavedienu skaits. Tas nozīmē to, ka lielo atmiņas izmantošanu daļēji var novērst ar servera pavedienu skaita samazināšanu. Tomēr, lai tīmekļa serveris spētu vienlaicīgi apstrādāt vairākus pieprasījumus, būtu jāatrod balanss starp pavedienu skaitu un izmantoto atmiņu, jo tikai ar diviem pavedieniem lietotnes apmierinošu darbību nodrošināt nevar.

Bija jāsecina, ka pētāmajai lietotnei problēmas ir nevis ar atmiņas noplūdēm, bet atmiņas burbuļiem. Atmiņas burbuļa problēma pierādījās, atklājot funkcionalitāti, kuras bieža darbināšana noveda pie lielāka atmiņas patēriņa, nekā serverim pieejams. Darbinot šo funkcionalitāti, atmiņas patēriņš nepieauga bezgalīgi un pēc pietiekama pieprasījumu skaita izlīdzinājās. Tas pierādīja, ka lietotnei ir problēmas ar atmiņas burbuļiem.

Otra problēma bija ar pārāk lielu servera pavedienu skaitu. Servera uzstādījumos samazinot pavedienu skaitu, atmiņas patēriņš arī samazinājās.

Iezīmējās vairāki problēmas risināšanas varianti. Pirmais un vienkāršākais variants ir palielināt serverim pieejamo atmiņu. Tā kā atmiņas noplūdes nav konstatētas, ar lielāku atmiņu iespējams panākt lietotnes stabilu darbību, tomēr tas palielinātu servera uzturēšanas izmaksas.

Otrā iespēja ir samazināt servera pavedienu skaitu, tādējādi samazinot maksimālo atmiņas patēriņu. Kā negatīva blakne tam būtu servera spēja paralēli apstrādāt vairākus pieprasījumus.

Trešā darbība, ko varētu veikt servera darbības uzlabošanai, ir lietotnes atmiņas patēriņa optimizācija. Funkcionalitāte, kas tika atklāta kā liela atmiņas patērētāja, ir pasūtījumu saraksta atveidošana ar datiem no vairākām saistītām datu bāzes tabulām. Optimizējot datu bāzes pieprasījumu veidošanu un līdz ar to arī lieku datu ievietošanu kešatmiņā, varētu panākt ievērojamu atmiņas patēriņa samazināšanos.

Lai nebūtu jātērē līdzekļi papildu atmiņai serverim, problēmu varētu risināt ar otrās un trešās iespējas apvienojumu. Optimizējot apskatīto funkcionalitāti, var panākt atmiņas lietojuma samazinājumu. Tomēr, lai vēl vairāk nodrošinātos pret atmiņas pārpildīšanos cita atmiņas burbuļa gadījumā, varētu samazināt arī servera pavedienu skaitu.

REZULTĀTI

Sākotnēji veiktie testi uz atsevišķas darbstacijas nedeva gaidītos rezultātus. Iegūtie dati atšķīrās no tiem, kas vēlāk tika iegūti no veiktajiem testiem produkcijas vidē.

Pētījuma rezultātā tika konstatēts, ka pētāmās lietotnes problēma ir nevis atmiņas noplūdes, bet gan atmiņas burbuļu veidošanās. Produkcijas vidē veikto testu rezultāti noraidīja atmiņas noplūdes un izgaismoja neefektīvu lietotnes atmiņas patēriņu. Pētījuma sākumā izteiktā hipotēze par to, ka pētāmajā lietotnē ir atmiņas noplūdes, ir jānoraida. Nevar garantēt to, ka lietotnē vispār nav atmiņas noplūdes, bet konstatēto problēmu izraisa atrastais atmiņas burbulis, kas tika noturēts par atmiņas noplūdi. Problēmas atrisināšanai tika noteikti vairāki iespējamie scenāriji, un iegūts aptuvens tālākās darbības plāns.

SECINĀJUMI

Tīmeklī ir pieejama informācija par to, kā meklēt ar atmiņu saistītas problēmas Ruby lietotnēs, bet viena konkrēta metode nav. Tika izmēģināti vairāki atrastie rīki un metodes atmiņas noplūžu konstatēšanai, bet tie bija domāti jau kādas konkrētas funkcionalitātes pārbaudīšanai. Pētāmās lietotnes gadījumā vajadzēja lokalizēt atmiņas problēmas izraisītāju. Tāpēc nederēja akli mēģinājumi uz labu laimi atrast vainīgo vietu. Palīdzēja servera žurnāla salīdzināšana ar atmiņas noslodzes grafiku, lai varētu atrast atmiņas aizpildīšanās izraisītāju. Šāda metode nebija aprakstīta tīmeklī atrodamajā informācijā.

Pētījumā izvirzītā hipotēze par atmiņas noplūžu klātbūtni ir noraidīta. Pētījuma rezultātā ir noskaidrots, ka pie atmiņas pārslodzes ir vainojama atmiņas burbuļa veidošanās un tātad neefektīva koda darbība, kā arī pārāk lielais servera pavedienu skaits.

Problēmas iespējamie risinājumi ir pieejamās atmiņas palielināšana serverim, pavedienu skaita samazināšana un lietotnes darbības optimizācija. Visticamāk atmiņas pārpildīšanās novēršanai tiks veikta lietotnes darbības optimizācija, kā arī pavedienu skaita samazināšana.

Pētījuma turpinājumā varētu veikt ilgstošu produkcijas servera atmiņas noslodzes un žurnāla analīzi, lai atrastu citus iespējamos atmiņas burbuļus.

IZMANTOTĀ LITERATŪRA UN AVOTI

1. P. Shaughnessy. *Ruby Under a Microscope*, 2013, 298–301. lpp.
2. "Understanding how Ruby stores objects in memory – the Ruby Heap," emuārs, 29.10.2009; http://www.theirishpenguin.com/2009/10/29/understanding-how-ruby-stores-objects-in-memory-the-ruby-heap.html.
3. T. Ball, "Watching and Understanding the Ruby 2.1 Garbage Collector at Work," emuārs, 12.03.2014; https://thorstenball.com/blog/2014/03/12/watching-understanding-ruby-2.1-garbage-collector/.
4. R. Hastings, B. Joyce, "Purify: Fast detection of memory leaks and access errors," *Proc. of the Winter 1992 USENIX Conference*, 1991.
5. E. A. Gupta, "Ruby 2.1: RGenGC," emuārs, 30.12.2013; http://tmm1.net/ruby21-rgengc/.
6. "Class : Symbol – Ruby 2.2.0", 01.04.2017; http://ruby-doc.org/core-2.2.0/Symbol.html.
7. R. Schneeman, "Symbol GC in Ruby 2.2," emuārs, 19.01.2015; https://www.sitepoint.com/symbol-gc-ruby-2-2/.
8. R.Schneeman, "Debugging a Memory Leak on Heroku," emuārs, 26.02.2017; https://blog.codeship.com/debugging-a-memory-leak-on-heroku/.
9. R. Schneeman, "Puma Worker Killer"; https://github.com/schneems/puma_worker_killer.
10. R. Schneeman, "Derailed Benchmarks"; https://github.com/schneems/derailed_benchmarks.
11. L. Simoneau, "Hunting Down Memory Issues in Rails," emuārs, 11.09.2015; http://rea.tech/hunting-down-memory-issues-in-rails/.
12. ScoutApp, "Debugging memory bloat," 18.07.2016; http://book.scoutapp.com/memory-bloat.html.
13. B. Marzolf, "Hunting Down Memory Issues In Ruby: A Definitive Guide,"; https://www.toptal.com/ruby/hunting-ruby-memory-issues.
14. C. Dryden, "How to debug Ruby memory issues," emuārs, 16.09.2015; http://eng.rightscale.com/2015/09/16/how-to-debug-ruby-memory-issues.html .
15. ASoftCo, "leaky-gems"; https://github.com/ASoftCo/leaky-gems.
16. J. Isip, "Puma – A modern concurrent web server for Ruby"; http://puma.io.
17. BlazeMeter. "The Load Testing Cloud"; https://chrome.google.com/webstore/detail/blazemeter-the-load-testi/mbopgmdnpcbohhpnfglgohlbhfongabi.
18. J. Isip, "puma/puma: A ruby web server built for concurrency"; https://github.com/puma/puma#thread-pool.