LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

DALĪTA SISTĒMA ALGORITMISKU UZDEVUMU TESTĒŠANAI

BAKALAURA DARBS

Autors: **Aigars Eglājs**
Studenta apliecības Nr.: ae09004
Darba vadītājs: Dr.sc.comp. Guntis Arnicāns

RĪGA 2013

ANOTĀCIJA

Bakalaura darbā ir pētīta algoritmisku uzdevumu testēšana dalītā sistēmā un no tā izrietošo problēmu risinājumi.

Programmu testēšana aizņem daudz resursu, kuri nepieciešami tikai noteiktos laiku posmos, un bieži vien to precīzs skaits nav zināms iepriekš. Ne visi serveri dod vienādus un atkārtojamus rezultātus.

Bakalaura darbā tiek apskatīta testēšanas sistēmas izstrāde, kas dod atkārtojumus rezultātus neatkarīgi no sistēmas noslodzes un ir droša testējot nezināmas izcelsmes C, C++ un PASCAL programmas. Šādai sistēmai jāspēj palielināt vai samazināt resursu apjomu atkarībā no sistēmas noslodzes. Izpētīti šādas sistēmas izveidošanas risinājumi un identificētas iespējamās problēmas.

Darba gaitā tika izstrādāta testēšanas sistēma, kas tika izmantota Latvijas 26. informātikas olimpiādes 1. un 2. posmā.

Atslēgvārdi: automatizēta testēšana, mērogošana, smilškaste, instrukciju mērīšana

ABSTRACT

This Bachelor's thesis studied algorithmic problem testing in distributed system and the resulting challenges.

Program testing takes a lot of resources and these resources are only required in specific time frames, and often the exact number of resources is not known in advance. Not all servers give equal and reproducible results.

This paper shows testing system design, which gives reproducible results regardless of system load and is safe for testing unknown C, C++ and PASCAL programs. Such a system should be able to increase or decrease amount of available resources depending on system load. This paper analyses possible problems and various ways of creating such a system.

Result of this paper is a testing environment that was used in 1st and 2nd round of the 26th Latvian Olympiad in Informatics.

Keywords: automated testing, scaling, sandbox, instruction counting

SATURS

Anotācija ... 2

Abstract ... 3

Ievads ... 6

1. Algoritmisku uzdevumu sacensības ... 8

  1.1. Algoritmiski uzdevumi ... 8

  1.2. Sacensības ... 8

  1.3. Sacensību tipi ... 9

  1.4. Pašreizējā situācija ... 9

2. Risinājumu vērtēšana ... 11

  2.1. Risinājuma atbildes novērtēšana ... 11

  2.2. Risinājuma sarežģītības novērtēšana ... 11

  2.3. Programmas instrukciju skaitīšana ... 15

  2.4. Izvēlētais risinājums ... 16

3. Drošības nodrošināšana ... 20

  3.1. Uzbrukuma vektori ... 20

  3.2. Linux piedāvātie risinājumi ... 21

  3.3. Izvēlētais risinājums ... 26

4. Sistēmas arhitektūra ... 32

  4.1. Testēšanas sistēmas arhitektūra un darbība ... 32

  4.2. Testēšanas sistēmas lietotne ... 33

  4.3. Testēšanas virsotnes ... 33

  4.4. Virsotņu menedžeris ... 33

5. Eksperimenti mākonī ... 38

  5.1. Eksperimenti ar konstantu serveru skaitu ... 39

  5.2. Eksperimenti ar mainīgu serveru skaitu ... 41

6. Noslēgums ... 44
6.1. Rezultāti ... 44
6.2. Nākošie soļi ... 44
Izmantotā literatūra ... 46
Pielikumi ... 48
1. Pielikums ... 48
2. Pielikums ... 49

IEVADS

Programmēšanas sacensības ir sacensības, kurās dalībnieki risina algoritmiskus uzdevumus noteiktās laika robežās. Programmēšanas sacensībās bieži vien ir problēmas ar iesūtīto risinājumu ātru testēšanu un rodas garas rindas iesūtīto risinājumu apstrādē. Šīs rindas ietekmē lietotāju rezultātus, jo lietotājs nevar būt pārliecināts, ka uzdevums ir atrisināts pareizi pirms nav saņemta atbilde no sistēmas. Pareiza resursu rezervēšana un sadalīšana ļauj gan paātrināt sistēmas darbību, gan ieekonomēt uz serveru izmaksām.

Šādu problēmu var vispārināt uz jebkādu programmu testēšanu un resursu piešķiršanas problēmu, kurai svarīgs ir testēšanas ātrums un resursu optimāls sadalījums. Katru testējamu programmu var uztvert kā algoritmisku uzdevumu un testpiemērus, ar kādiem šī programma tiek testēta. Arī vispārinātām testējamām programmām eksistē atmiņas, laika un drošības ierobežojumi, kurus var pielīdzināt algoritmisku uzdevumu ierobežojumiem.

Tā kā vienam uzdevumam var būt vairāki desmiti testpiemēri un katra testpiemēra notestēšana var aizņemt vairākas sekundes, tad uzdevuma risinājumu testēšana var kļūt par ļoti laikietilpīgu procesu. Algoritmisku uzdevumu testpiemēri ir savā starpā nesaistīti, tāpēc tos ir iespēja izpildīt paralēli uz vairākām sistēmām, lai paātrinātu kopējo testēšanas laiku. Tā kā testēšana notiek tikai pēc jaunu risinājumu saņemšanas, tad var eksistēt vairāki laika periodi, kad netiek izmantoti visi datorresursi. Šī iemesla dēļ tiek piedāvāts risinājumā izmantot mākoņdatošanu. Tas dod iespēju ātri papildināt testēšanas sistēmu ar papildus resursiem vai atteikties no šiem resursiem, ja tie vairs nav vajadzīgi. Savā darbā tuvāk apskatīšu šādas sistēmas realizāciju.

Lai attīstītu un novērtētu savas spējas risināt algoritmiskus uzdevumus, programmētāji piedalās dažādās sacensībās. Šādu sacensību rezultātā programmētāji iegūst jaunas zināšanas, prestižu un pat dažādas balvas. Tādēļ sacensību norisei jābūt bez problēmām un vēlams atbildes uz risinājumiem sniegt ātri, jo šī atbilde var ietekmēt sacensību dalībnieka turpmāko darbību.

Tā kā datoru procesori kļūst arvien ātrāki, tad precīza risinājuma sarežģītības novērtēšana prasa arvien lielākus datu apmērus. Liela izmēra testpiemēri liek lielāku uzsvaru uz datu nolasīšanas ātrumu nekā apstrādes ātrumu. Šī iemesla dēļ arvien lielāka ietekme ir diska un procesora kešatmiņai, kas rada lielākas kļūdas precīzā risinājuma laika noteikšanā. Tiek piedāvāta programmu instrukciju skaitīšana kā risinājums šai problēmai. Šis risinājums un iesūtījumu funkcionālā novērtēšana ir aprakstīta 2. nodaļā.

Tā kā tiek izpildīti nezināmas izcelsmes pirmkodi un mērķauditorija ir programmētāji, var gadīties, ka sistēma tiek izmantota ļaunprātīgi, tādēļ īpaša uzmanība jāpievērš sistēmas drošībai, kas tiek apskatīta 3. nodaļā.

Lai nodrošinātu ātras atbildes uz algoritmu risinājumiem nepieciešami liels skaits datorresursu. Šie resursi ir nepieciešami tikai noteiktā laika posmā, bet ideālais resursu skaits ir grūti nosakāms, ja nav zināms precīzs testēšanas grafiks, un tāpēc jābūt iespējai tos īsā laikā papildināt vai noņemt, ja tie netiek izmantoti. Šī resursu mērogojamība tiks apskatīta 4. nodaļā.

5. nodaļā tiks tuvāk apskatīta dažādu risinājumu izpilde uz Latvijas 26. informātikas olimpiādes 2. posma iesūtījumiem.

Darba noslēgumā tiks izvērtēts izmantotais risinājums un apskatīti nākamie pētāmie soļi un idejas tālākai sistēmas attīstībai.

1. ALGORITMISKU UZDEVUMU SACENSĪBAS

1.1. Algoritmiski uzdevumi

Algoritmiski uzdevumi ir tādi programmēšanas uzdevumi, kuros lietotājam ir prasīts atrisināt kādu konkrētu problēmu, piemēram, atrast n-to pirmskaitli. Uzdevumiem var būt viena vai vairākas pareizas atbildes. Uzdevumi sastāv no apraksta, ievaddatiem, izvaddatiem un dažādiem ierobežojošiem faktoriem – izpildes laika, atļautās atmiņas lieluma un ievaddatu izmēriem. Ievaddati tiek iegūti no faila vai standarta ieejas. Var būt uzdevumi, kas ir interaktīvi, tas ir, atkarībā no programmas darbības var mainīties ievaddati, piemēram, spēle, kur jāuzmin skaitlis, un ievaddatos tiek dots vai programmas minētais skaitlis ir mazāks vai lielāks par meklējamo skaitli.

Katram uzdevumam var būt vairāki testi. Testi parasti tiek izveidoti tā, lai nosegtu dažādas uzdevuma nianses un varētu atšķirt, kāds algoritms tiek izmantots uzdevuma risinājumā. Uzdevuma testi parasti tiek veidoti ar mērķi, ka algoritmi, kam izpildes laiks vai atmiņas patēriņš ir lielāks par vēlamo algoritmu, nespēj sniegt pareizu atbildi uz visiem testiem noteiktajās izpildes laika un atmiņas robežās.

Kā vienkāršs algoritmiska uzdevuma piemērs ir skaitļa sadalīšana reizinātājos. Ievaddatos ir dots skaitlis, kas jāsadala reizinātājos, un programmai jāizvada visi skaitļa reizinātāji. Uzdevuma risinājumam jādod atbilde sekundes laikā, neizmantojot vairāk kā 32MB.

Daži algoritmisku uzdevumu piemēri un to risinājumi ir parādīti pielikumā Nr. 1.

1.2. Sacensības

Pastāv dažādas programmēšanas sacensības, kurās var risināt dažādus algoritmiskus uzdevumus noteiktos laika posmos. Šīs sacensības notiek testēšanas sistēmā, kurā tiek vērtēti sacensību dalībnieku iesūtītie uzdevumi. Pastāv testēšanas sistēmas, kuras neatbalsta sacensības vai atbalsta uzdevumu risināšanu ārpus sacensībām. Šādas sistēmas var uztvert par sistēmām, kas atbalsta sacensības, kas ilgst bezgalīgi ilgi.

Vidusskolēniem ir informātikas olimpiādes Latvijas mērogā un internacionālā mērogā - Latvijas informātikas olimpiāde (LIO)[1], Baltijas informātikas olimpiāde (BOI), Centrāleiropas informātikas olimpiāde (CEOI)[2], Internacionāla informātikas olimpiāde (IOI)[3] un citas. Studentiem galvenās sacensības ir ACM internacionālās studentu programmēšanas sacensības

[1] http://vip.latnet.lv/lio/
[2] http://en.wikipedia.org/wiki/Central_European_Olympiad_in_Informatics
[3] http://www.ioinformatics.org/

(ACM-ICPC)[4]. Šīs sacensības ir klātienes sacensības un ilgst 1 vai 2 dienas. Katrā dienā ir rezervētas 5 stundas 3 līdz 11 uzdevumu risināšanai.

Ir pieejamas arī ļoti daudzas sacensības, kas notiek tiešsaistē. Populārākās tiešsaistes sacensības ir Horvātu atklātās informātikas olimpiādes (COCI)[5], ASV informātikas olimpiādes (USACO)[6], *TopCoder*[7] sacensības, *CodeForces*[8] sacensības.

1.3. Sacensību tipi

Pastāv dažādi sacensību tipi. Šie tipi atšķiras pēc uzdevumu tipiem, to atrisināšanas iespējām un risinātāju komandas sastāva.

Skolēnu informātikas olimpiādes parasti ir individuālas sacensības, kurās uzdevumus var atrisināt arī daļēji. Šajās sacensības pieejamo programmēšanu valodu skaits ir stipri ierobežots, pašlaik tās ir C, C++ un PASCAL. Šajā darbā ir aprakstīta testēšanas sistēmas izveide tieši šim mērķim – skolēnu sagatavošanai informātikas olimpiādēm.

ACM-ICPC ir komandas sacensības. Komandā ir 3 cilvēki un pieejams tikai viens dators, tātad jāpievērš uzmanība ne tikai uzdevuma risināšanai, bet arī pareizai resursu sadaļai, tas ir, kurš no komandas dalībniekiem izmanto datoru, lai rakstītu uzdevuma risinājumu. Šajās sacensībās uzdevumi ir jāatrisina pilnīgi, tas ir, uz visiem uzdevuma piesaistītajiem testiem jādod pareiza atbilde, iekļaujoties izpildes laika un atmiņas robežās, daļēji uzdevuma risinājumi nepastāv.

Pastāv arī sacensības, kurās ir iespēja dalībniekiem meklēt kļūdas citu dalībnieku pirmkodos. Ja dalībniekiem šķiet, ka ir atradis kļūdu cita dalībnieka kodā, tad viņš var padot ievaddatus, uz kuriem programma dod nepareizu atbildi.

Testēšanas sistēmas darbībā sacensību tips nav svarīgs. Dažādos sacensību tipus var simulēt tīmekļa serveris, attēlojot rezultātus dalībniekiem.

1.4. Pašreizējā situācija

Tā kā algoritmiskas sacensības notiek noteiktos laikos un bieži vien ilgst tikai pāris stundas, tad pareizu resursu sadalīšana ir svarīga daļa no šo sacensību administrācijas. Lielāko daļa no laika datorresursi ir dīkstāvē, tādēļ datoru vai serveru pirkšana un īrēšana uz visu laika posmu nav ekonomiski izdevīga. Arī precīza nepieciešamo resursu skaita noteikšana ir gandrīz neiespējama.

[4] http://icpc.baylor.edu/
[5] http://www.hsin.hr/coci/
[6] http://usaco.org/
[7] http://www.topcoder.com/
[8] http://codeforces.com/

Ar precīzu resursu skaitu šeit domāta sistēma, kurai nav lieku resursu un to netrūkst visā sacensību laika posmā. Pašlaik lielākā daļa testēšanas sistēmas ir veidotas no atveltītiem datorresursiem, kas nav ekonomiski izdevīgi. Ja rezervēto datorresursu skaits ir mazāks nekā nepieciešams, tad rodas rindas un iesūtījumu apstrāde var ieilgt, bet ja rezervēto resursu skaits ir lielāks par nepieciešamo, tad resursi var pavadīt lielu laika daļu dīkstāvē, un šie resursi rada papildus nevajadzīgas izmaksas.

Kā viens no risinājumiem ir šādas sistēmas izveide mākonī, kur jaunu resursu pieprasīšana vai atteikšanās ir vienkāršs un ātrs process.

2. RISINĀJUMU VĒRTĒŠANA

Testēšanas sistēmas galvenais uzdevums ir novērtēt iesūtītos risinājumus un dot atbildi par to rezultātiem. Vienkāršotais variants ir noteikt, vai iesūtītā risinājuma dotā atbilde uz konkrētiem ievaddatiem ir pareiza vai nepareiza. Lietotāja algoritms var darboties pareizi, bet ja tas dod atbildi pārāk lēni vai izmanto pārāk daudz atmiņas, tad arī šādi risinājumi pieskaitāmi pie nepareiziem.

2.1. Risinājuma atbildes novērtēšana

Ja risinājums ir apstrādājis ievaddatus un devis atbildi noteiktā laika limitā, un nav pārtērējis atļauto atmiņas daudzumu, tad šī lietotāja atbilde tiek salīdzināta ar gaidāmo atbildi. Atkarībā no uzdevuma šī atbildes salīdzināšana var būt lietotāja un uzdevuma autora atbilžu salīdzināšana simbolu pa simbolam vai lietotāja un uzdevuma autora atbilžu salīdzināšana, ignorējot atstarpes un jaunas līnijas. Ja uzdevumam var būt vairākas pareizas atbildes vai atbilde var būt ar kļūdu, piemēram, reālu skaitļu atbildes, tad var tikt izmantota arī uzdevumam pielāgota pārbaudes programma. Šī pārbaudes programma kā argumentus saņem testpiemēra ievaddatus, lietotāja atbildi un testpiemēra sagaidāmo atbildi, ja tāda ir. Tad šī pārbaudes programma pārbauda vai lietotāja atbilde ir derīga attiecīgajiem ievaddatiem.

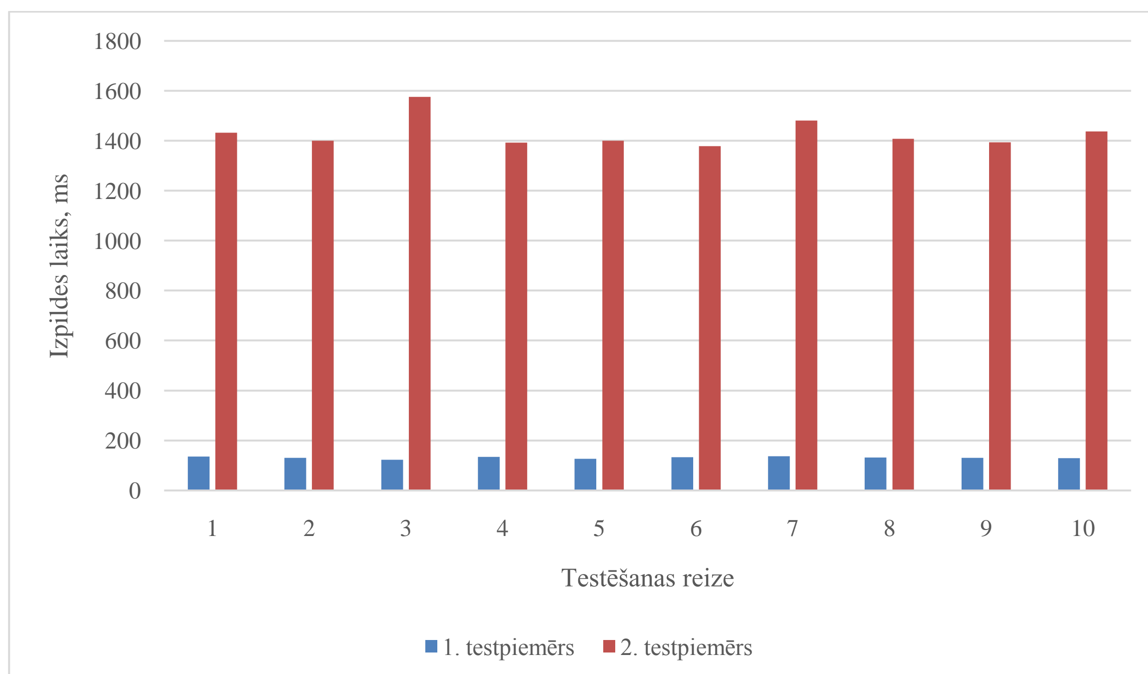
2.2. Risinājuma sarežģītības novērtēšana

Risinājuma sarežģītība tiek vērtēta ar programmas izpildes laika un patērētās atmiņas palīdzību. Praktiski risinājuma sarežģītība tiek novērtēta palaižot programmu uz noteiktiem ievaddatiem. Diemžēl viena risinājuma vairākkārtēja palaišana var dot atbildi atšķirīgos ātrumos dažādu iemeslu dēļ, piemēram:

- Mašīna, uz kuras tiek palaists risinājums, tiek kādā brīdī noslogota ar citām programmām vai darbībām;
- Var tikt neiztīrīta atmiņa, piemēram, kešoti daži rezultāti;
- CPU optimizācijas un kontekstu pārslēgšana;
- Atmiņas un diska fragmentācija.

Tika apskatīts uzdevums, kurā jānolasa skaitļi no faila un jāatrod šo skaitļu mediāna. Pirmajā testpiemērā ir doti 10^6 skaitļi un otrajā testpiemērā ir doti 10^7 skaitļi. Testējamā programma tika rakstīta C++ valodā, tā nolasīja skaitļus no faila, tos sakārtoja un izvadīja šo skaitļu mediānu. Programma tika palaista uz Linux vides un uz datora, kas papildus neveica citas

resursietilpīgas darbības. Visas šajā nodaļā testētās programmas tiek palaistas uz šīs pašas vides, ja netiek norādīts savādāk.



2.1. att. **Programmas izpildes laiki uz vienas mašīnas**

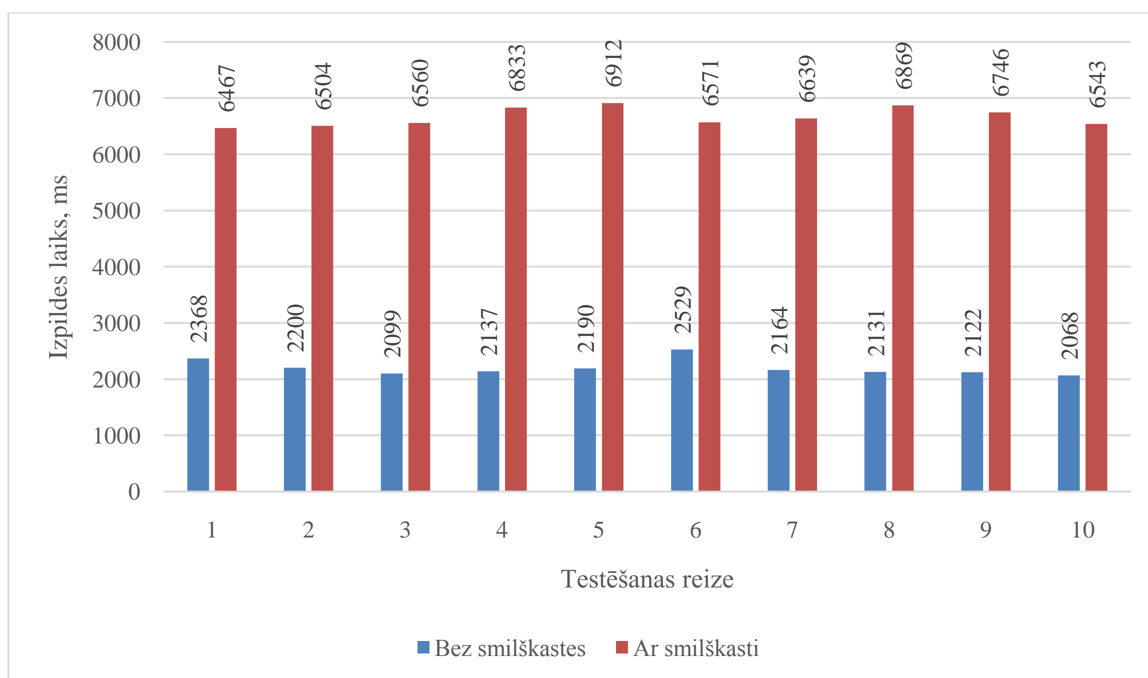
Attēlā 2.1. ir redzams šīs programmas divu testpiemēru vairākas palaišanas reizes uz vienas mašīnas. Pirmajam testpiemēram izpildes laiks svārstās no 1378 līdz 1576, otrajam testpiemēram izpildes laiks svārstās no 123 līdz 137. Tātad kļūda precīzam izpildes laika mērījumam var būt līdz pat 15%. Līdzīgas izpildes laika kļūdas apskatītas *M. Mareš* darbā [1].

Tā kā sistēmā tiek testēti iesūtījumi ar nezināmas izcelsmes kodu un var saturēt darbības, kas var apdraudēt sistēmu, tad programmas nepieciešams testēt *smilškastē*. Smilškaste ir drošības mehānisms, kas ierobežo programmas atļautās darbības. Plašāk smilškastes mehānisms tiks apskatīts 3.3. nodaļā. Bieži vien šī ierobežošana notiek aktīvi filtrējot dažādas programmas darbības, tādēļ smilškastes izmantošana var radīt lielu izpildes laika un atmiņas virstēriņu.

Tika apskatīta programmas izpildes laika atšķirības to palaižot smilškastē un bez smilškastes. Šī programma ir risinājums 2013. gada Baltijas informātikas olimpiādes uzdevumam „Pēdas sniegā”[9], un tā tika testēta uz viena no oficiālajiem testpiemēriem.

---

[9] http://boi2013.informatik-olympiade.de/wp-content/uploads/2013/04/tracks-LVA.pdf

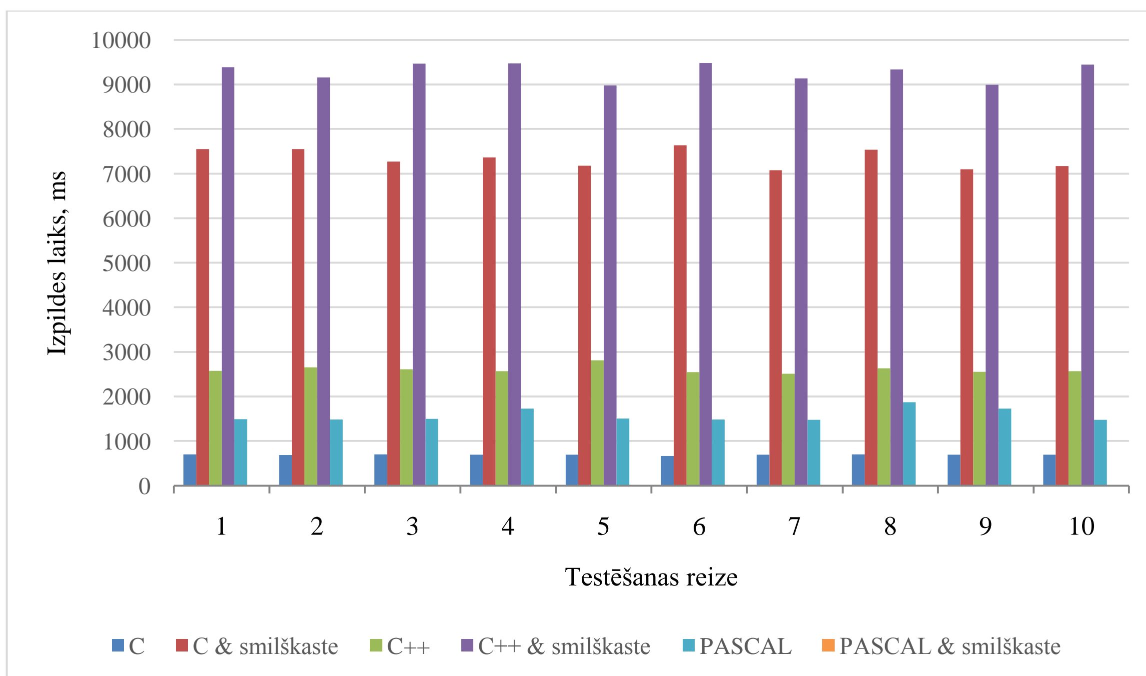


2.2. att. **Programmas izpildes laiki ar smilškasti un bez smilškastes**

Attēlā 2.2. redzama šīs programmas atšķirība izpildes laikam, kad tā tiek palaista izvēlētā risinājuma smilškastē. Atkarībā no sistēmas izsaukumu skaita smilškastes virstēriņš var būt gan nenozīmīgs, gan palēnināt programmas darbību par vairākām kārtām. Darbības, kas visvairāk ietekmē smilškastes virstēriņu, ir lasīšanas un rakstīšanas darbības, jo tās sevī ietver daudzus sistēmas izsaukumus, tādēļ uzdevumos, kuros ievaddatu un izvaddatu daudzums ir mazs, smilškastes virstēriņš ir mazs. Šī iemesla dēļ vēlams sastādīt uzdevumus, kur uzsvars ir uz algoritmu nevis uz datu ielasīšanu un izvadi. Diemžēl bieži vien datu apjoms, kas nepieciešams risinājumu testēšanai, var būt liels, piemēram, testpiemērs šai testēšanai bija 16 megabaitu liels.

Atšķirīgi ir arī dažādu valodu ielasīšanas ātrumi, kas dod priekšroku dažu programmēšanas valodu lietotājiem. Attēlā 2.3. redzams C, C++ un PASCAL valodā rakstītu programmu, kas ielasa 3 000 000 skaitļus no faila. C valodā tika izmantots *scanf* [10] metode, C++ valodā izmantotā metode bija *cin* [11] un PASCAL tika izmantota *readln* [12].

[10] Standarta ielasīšanas metode C valodā - http://www.cplusplus.com/reference/cstdio/scanf/
[11] Standarta ielasīšanas metode C++ valodā - http://www.cplusplus.com/reference/iostream/cin/
[12] Standarta ielasīšanas metode PASCAL valodā - http://www.freepascal.org/docs-html/rtl/system/readln.html



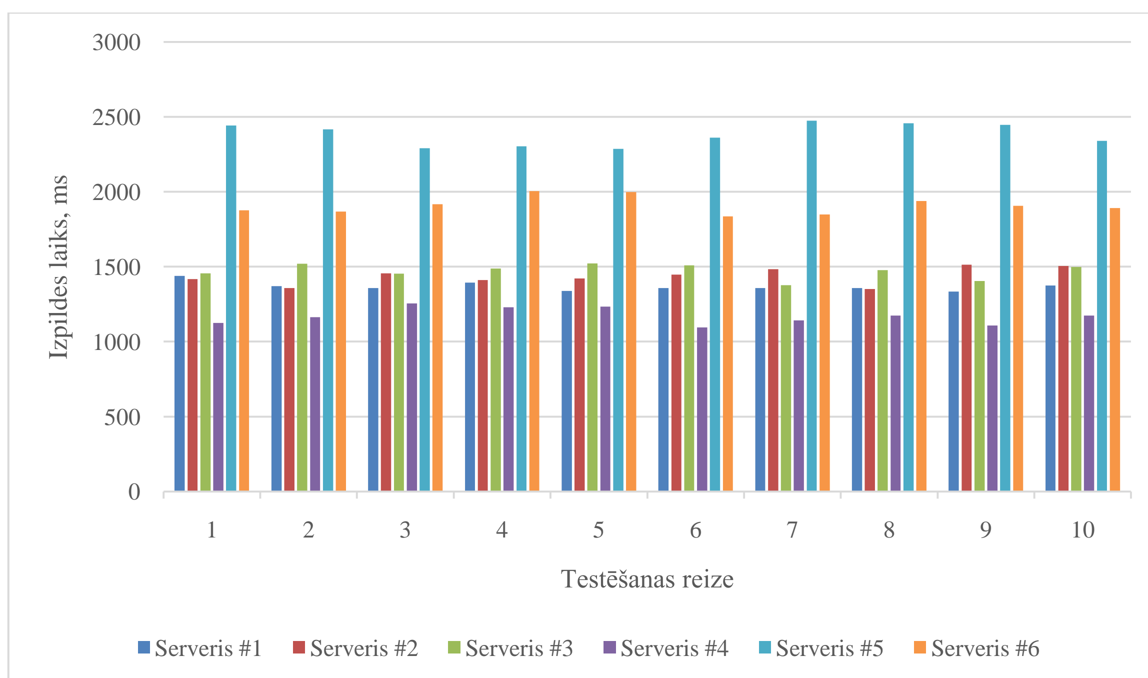
2.3. att. **Dažādu valodu ielasīšanas ātrums**

Attēlā 2.3. redzams šo programmu izpildes laiku salīdzinājums. Kad PASCAL programma tika palaista smilškastē, smilškastes virstēriņš palēnināja programmas darbību tik tālu, ka programma nespēja nolasīt visus datus noteiktajā laika limitā – 30 sekundēs. Tas ir izskaidrojams ar PASCAL lasīšanas buferizāciju, kuras dēļ tiek veikti papildus sistēmas izsaukumi katru reizi, ielasot datus no faila.

Piedāvātais risinājums ietver programmu testēšanu uz mākoņa platformas. Viena no mākoņdatošanas īpašībām ir vairāku neatkarīgu sistēmu virtualizācija zem vienas fiziskas sistēmas. Šīs īpašības dēļ testēšanai paralēli var parādīties negaidītas veiktspējas problēmas, piemēram, konteksta pārslēgšanās un diska fragmentācija [2]. Šīs veiktspējas problēmas var ietekmēt programmu izpildes laikus un dot ļoti atšķirīgus rezultātus dažādos serveros, kaut gan servera arhitektūra ir vienāda.

Tika apskatīta iepriekš minētā programma, kas nolasa skaitļus un atrod mediānu, uz sešiem vienādi izveidotiem serveriem mākonī. Šie seši serveri tika izvēlēti no mākoņdatošanas servisa piedāvātājā *Rackspace*[13] ar vienādiem servera parametriem – Linux operētājsistēma, 256 MB RAM un 2,1 GHz AMD procesors.

[13] http://www.rackspace.com/



2.4. att. **Programmas izpildes laiki uz vairākām mašīnām mākonī**

Attēlā 2.4. redzami šīs programmas vairāku testēšanas reižu izpildes laiki. Trīs no sešiem serveriem dod ļoti tuvus rezultātus (#1, #2, #3), bet citiem ir pat līdz sekundes nobīdei.

Serveru nespēja izpildīt vienādas programmas vienādos laikos uz dažādiem serveriem var radīt gadījumus, kad viena lietotāja risinājums netiek apstiprināts, bet cita lietotāja risinājums vai tā paša lietotāja atkārtots risinājuma iesūtījums tiek akceptēts kā pareizs risinājums. Tā kā laiki, kādā programma dod atbildi, var svārstīties, tad nav konkrētu pierādījumu, ka sistēmā tika testēts pareizais iesūtījums, jo nav iespējams atkārtot identisku rezultātu.

2.3. Programmas instrukciju skaitīšana

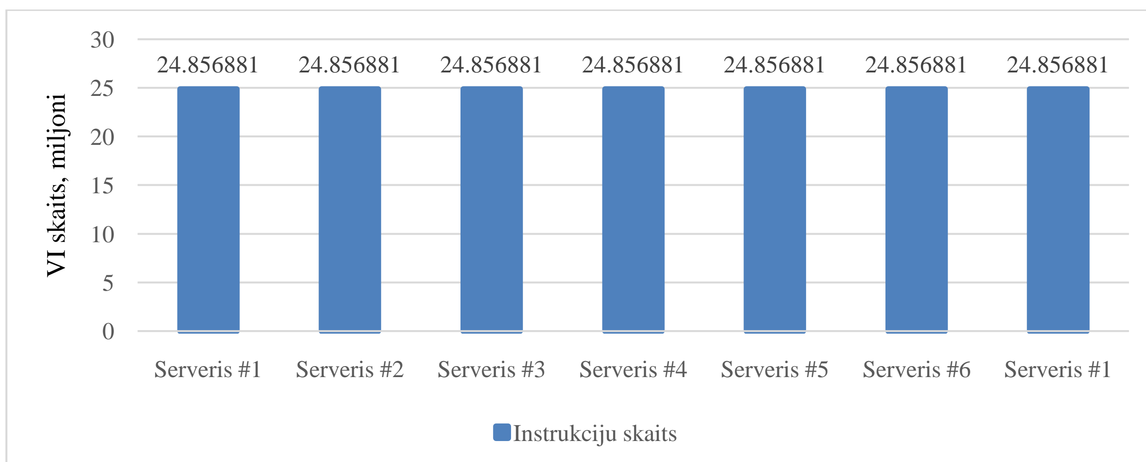
Varētu salīdzināt risinājumus, skaitot to izdarītās darbības. Par vienu darbību var tikt uzskatīta viena mašīnvalodas instrukcija, piemēram, reģistra nolasīšana, vērtības piešķiršana reģistram, matemātiska darbība un citas. Šāda darbību skaitīšana ir gandrīz neiespējama, veicot statisku koda analīzi, tādēļ tās būtu jāskaita programmai darbojoties uz noteiktiem ievaddatiem. Izpildīto darbību skaitu varētu novērtēt pēc programmas izpildes laika ņemot vērā sistēmas taktātrumu. Piemēram, ja serverim ir procesors ar 2 MHz taktātrumu un tas spēj izpildīt vienu vienkāršu darbību (piemēram, saskaitīšana un atņemšana) 10 taktīs, tad var pieņemt, ka tas spēj izpildīt 200 000 vienkāršas instrukcijas. Tātad, ja programmas izpilde aizņem vienu sekundi, tad varam pieņemt, ka programmas izpilde aizņēma apmēram 200 000 darbības. Dažādu procesora optimizāciju (zarojumparedzēšana, instrukciju paralēlā izpilde un citas) dēļ programma var

izpildīties ātrāk, tādejādi dodot ilūziju par ātrāku algoritmu, kaut arī šī pati programma uz cita veida procesora, kur šīs optimizācijas nav pieejams, varētu izpildīties ilgāk.

Viens veids kā iegūt identisku rezultātu, vairākas reizes testējot uz dažādām mašīnām ar dažādiem procesoriem, ir skaitīt programmas izpildītās instrukcijas. Viens no rīkiem, kas dod iespēju skaitīt programmas izpildītās instrukcijas, ir *valgrind*. [3]

Lai veiktu programmas profilēšanu, *valgrind* izmanto izjaukt-atkārtot principu. Programmas mašīnkods tiek pārvērsts *valgrind* instrukcijās (VI). Viena mašīnkoda instrukcija var būt vairākas *valgrind* instrukcijas, šīm pārvērstajām instrukcijām tiek pievienotas dažādas profilēšanas un simulācijas funkcijas, un tad *valgrind* instrukcijas, profilēšanas un simulācijas funkcijas tiek pārvērstas atpakaļ uz mašīnkodu. Palaižot šo apstrādāto mašīnkodu, tiek iegūts mērāmās programmas *valgrind* instrukciju skaits [3]. Tā kā programma tiek salīdzināta ar *valgrind* instrukciju skaitu, tad pat uz dažādiem procesoriem šis skaits varētu būt vienāds.

Tika atkārtota attēlā 2.4. apskatītās programmas izpilde ar *valgrind* uz tiem pašiem sešiem serveriem. Šoreiz tika apskatīts nevis izpildes laiks, bet *valgrind* instrukciju skaits.

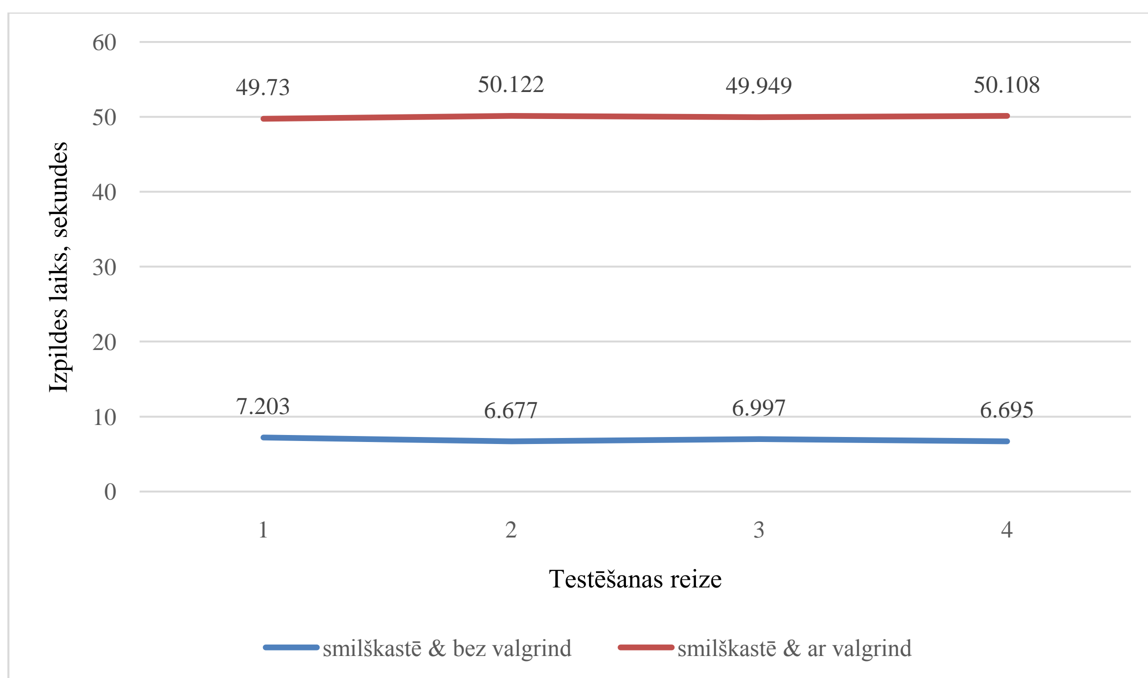


2.5. att. ***Valgrind*** **instrukciju skaits**

Attēlā 2.5. redzams šīs programmas instrukciju skaits uz 6 dažādiem serveriem un atkārtots programmas instrukciju skaits, palaižot programmu vēlreiz uz pirmā servera. Kā redzams instrukciju skaits šai programmai ir pilnīgi identisks un atkārtojams neatkarīgi no servera.

2.4. Izvēlētais risinājums

Tika apskatīts iepriekš minētās programmas „Pēdas sniegā" risinājums un izpildes laiks, palaižot smilškastē bez un ar *valgrind*.

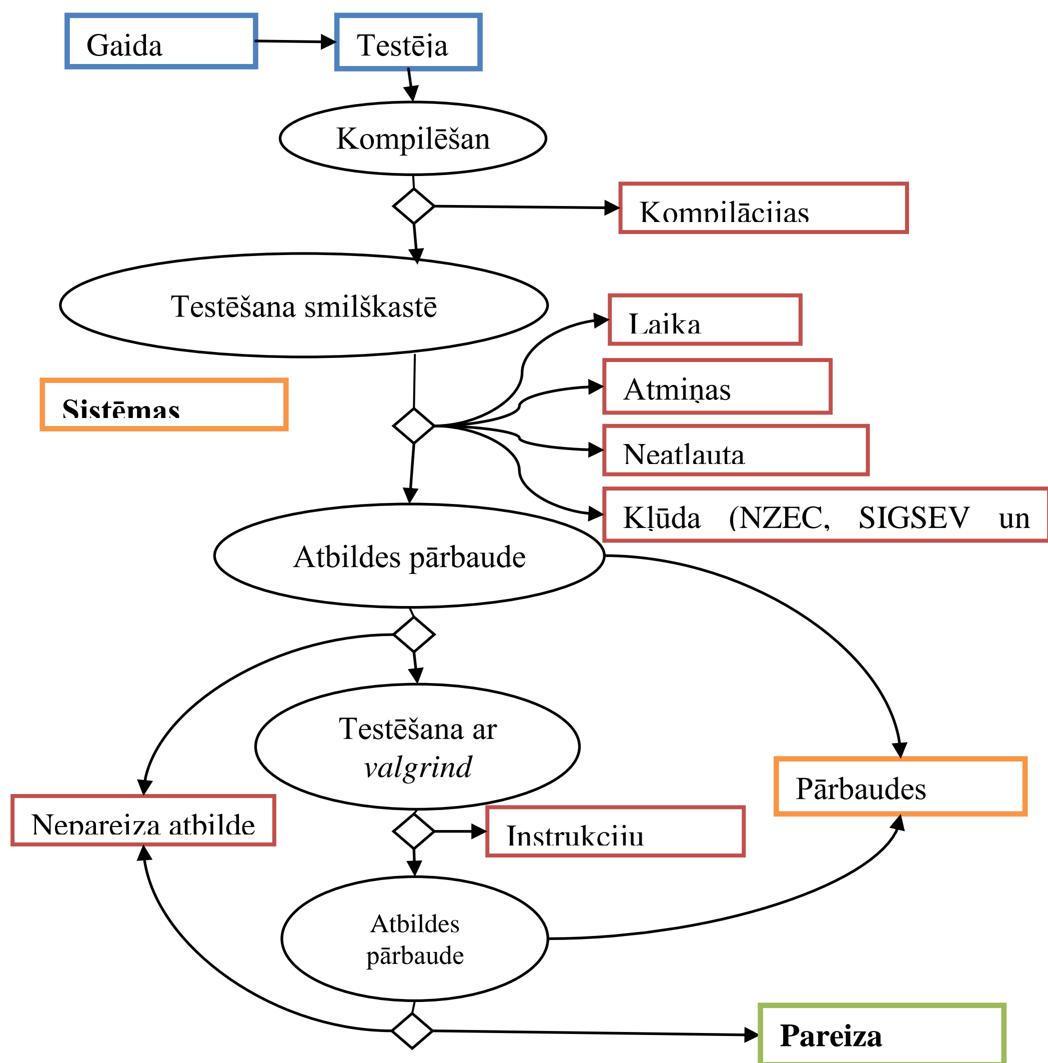


2.6. att. **Izpildes laiks**

Kā redzams attēlā 2.6. *valgrind* palielina šī piemēra izpildes laiku apmēram 7 reizes. Kā minēts *valgrind* dokumentācijā, tad instrukciju mērīšana var palēnināt programmas darbību līdz pat 100 reizēm.[14]

Izvēlētais risinājums ir atzīmēt nepareizās programmas, pirms tās tiek testētas ar *valgrind*. Smilškastes problēma ir precīza izpildes laika noteikšana, tāpēc, ja lietotāja risinājums nedod pareizu atbildi, pārtērē atmiņas daudzumu vai izmanto kādas neatļautas funkcijas, tad šis risinājums pavisam noteikti ir nepareizs un tādēļ pat netiek testēts ar *valgrind*. Lai arī nav iespējams noteikt precīzu izpildes laiku, ir iespējams noteikt aptuvenu izpildes laiku, tādēļ izpildes laika limits tiek uzstādīts divreiz lielāks nekā īstais, lai atļautu vietu izpildes laika kļūdai. Tādejādi, ja risinājums ir uz laika izpildes robežas, tad ir varbūtība, ka tas paspēs dot atbildi un tiks notestēts ar *valgrind*.

[14] http://valgrind.org/info/



2.7. att. **Testēšanas process**

Attēlā 2.8. redzama testēšanas procesa loģika un soļi. Ir divi pagaidu stāvokļi – „gaida rindā” un „testējas”. Ir septiņi neveiksmīgi rezultāti:

- Kompilācijas kļūda – lietotājam tiek parādīts kompilācijas ziņojums un visi iesūtījuma testi tiek izņemti no rindas;
- Laika limits – ir pārsniegts uzdevuma izpildes laika limits;
- Atmiņas limits – ir pārsniegts norādītais uzdevuma atmiņas limits;
- Neatļauta darbība – ir mēģināts izdarīt kādu neatļautu darbību, piemēram, nomainīt lietotāju, izveidot jaunu sakni, izdzēst vai piekļūt citiem failiem, vai jebkāda cita, iespējams, ļaunprātīga darbība;

- Kļūda – programma negaidīti pārtrauca darbību kļūdas dēļ, piemēram, mēģinot piekļūt neeksistējošai atmiņas adresei, vai dalīšana ar nulli;
- Nepareiza atbildi – novērtējot programmas doto rezultātu uz ievaddatiem tiek secināts, ka atbilde ir nepareiza;
- Instrukciju limits – programmas instrukciju skaits pārsniedz uzdevuma instrukciju limitu;

Iegūstot jebkuru no neveiksmīgiem rezultātiem tālāka programmas testēšana un apstrāde nenotiek.

Atbildes novērtēšana notiek ar papildprogrammas palīdzību, kura ieejā saņem testa ievaddatus, testa izvaddatus un lietotāja izvaddatus. Šīs papildprogrammas var būt vienkārša failu salīdzināšana, ņemot vērā tukšuma simbolus vai tos ignorējot, salīdzinot skaitļus ar pieļaujamu kļūdas līmeni, vai uzdevumam specifiska papildprogramma, kas var simulēt lietotāja atbildi. Ja uzdevuma izvaddati var būt neviennozīmīgi, tas ir, uz vieniem ievaddatiem pareizas var būt vairākas pareizas atbildes, tad pavisam noteikti ir vajadzīga uzdevumam specifiska papildprogramma. Viena uzdevuma testiem var būt arī atšķirīgas papildprogrammas.

Ir tikai viens derīgs rezultāta stāvoklis, un tas iestājas tikai pēc veiksmīgas *valgrind* testēšanas un veiksmīgas rezultāta pārbaudes.

Ir divi stāvokļi, kas ziņo par sistēmas kļūdu, - „pārbaudes kļūda" un „sistēmas kļūda". Pārbaudes kļūda iestājas, ja atbildes novērtēšanas papildprogramma nedod derīgu atbildi vai kādu iemeslu dēļ nestrādā. Sistēmas kļūda iestājas brīdī, kad kādā no testēšanas procesiem notiek kļūda vai nav iespējams izpildīt kādu no testēšanas darbībām. Ja iestājas pārbaudes kļūda, tad par to tiek paziņots administratoram. Ja iestājas sistēmas kļūda, tad testēšanas virsotne tiek atzīmēta kā neaktīva un turpmākas darbības tiek atļautas tikai pēc administratora pavēles. Sistēmas kļūdas gadījumā tiek automātiski pieprasīta arī jauna testēšanas virsotnes izveide.

Neviens no stāvokļiem neizraisa atkārtotu testa testēšanu. Atkārtot testēšanu var veikt manuāli tikai administrators. Tā ka iespējams sistēmas kļūdu ir izraisījis lietotāja iesūtījums, katra iesūtījuma pārtestēšana ir jānovērtē individuāli.

3. DROŠĪBAS NODROŠINĀŠANA

Viena no lielākajām problēmām šādās testēšanas sistēmās ir sistēmas drošība. Drošības problēmas rodas no vairākiem avotiem. Viens no galvenajiem iemesliem ir tas, ka sistēmā tiek palaistas lietotāja programmas, kuru darbība un mērķis nav iepriekš zināms. Šī iemesla dēļ jāpievērš liela uzmanība sistēmas drošībai.

3.1. Uzbrukuma vektori

Uzbrukuma vektori ir dažādi. Pastāv vispārēji uzbrukuma vektori – servera pārslogošana ar pieprasījumiem un dažādi tīmekļa drošības uzbrukuma veidi (XSS[15], SQL injekcijas[16], CSRF[17] un citi). Šos uzbrukuma veidus neapskatīšu tuvāk, jo tie nav izolēti testēšanas sistēmas drošības elementi. Tuvāk apskatīšu drošības aspektus, kas saistīti tieši ar programmu kompilēšanu, palaišanu un darbību.

Iespējamās programmu nenoteiktības dēļ iesūtīto pirmkodu statiska analīze nav pilnīgs veids kā nodrošināties pret ļaunprātīgu programmu darbību [4]. Šāda statiskā pirmkoda analīze toties var palīdzēt izfiltrēt vienkāršus uzbrukuma kodus un veikt citas noderīgas pirmkoda analīzes procedūras, piemēram, atzīmējot lietotajā pirmkoda klonus, lai noteiktu krāpšanās mēģinājumus [5]. Tā kā nav iespējams pilnīgi nodrošināties pret ļaunprātīgām darbībām ir nepieciešama aktīva aizsardzība pret ļaunprātīgu darbību.

Viena no lielākajām drošības problēmām ir dažādi *sistēmas izsaukumi*, kas varētu sabojāt sistēmu [6]. Sistēmas izsaukumi ir programmas instrukcijas, kas pārtrauc programmas darbību un nodod kontroli pārvaldošajai sistēmai. Šajā gadījumā tā būtu operētājsistēma. Diemžēl nav iespējams nobloķēt visus sistēmas izsaukumus, jo daži no tiem ir nepieciešami programmas darbībai. Situāciju sarežģītāku padara izsaukumi, kuriem ir svarīgs konteksts, tas ir, dažos gadījumos tiem jābūt atļautiem, dažos aizliegtiem. Piemēram, šādi izsaukumi ir darbības ar failiem. Viens no iespējamiem risinājumiem failu darbību izsaukumiem ir ievaddatus un izvaddatus iegūt no standarta ievades un izvades.

Bet failu darbības nav vienīgie sistēmas izsaukumi, kas rada sarežģījumus. Algoritmisko uzdevumu risināšanas sacensībās parasti nav atļauts izveidot jaunus pavedienus, lai samazinātu iespēju lietotājam palaist citas programmas vai radīt problēmas sistēmas izsaukumu filtrēšanā [7].

[15] *Cross Site Scripting* (https://en.wikipedia.org/wiki/Cross-site_scripting)
[16] http://en.wikipedia.org/wiki/SQL_injection
[17] *Cross Site Request Forgery* (https://en.wikipedia.org/wiki/Cross-site_request_forgery)

Aizliegta ir arī komunikācija ar tīklu, jo algoritmisku uzdevumu risināšanai nav nepieciešama saziņa starp dažādām mašīnām.

3.2. Linux piedāvātie risinājumi

Tā kā tika izvēlēts *valgrind* un tas oficiāli atbalsta galvenokārt tikai Linux, tad arī drošības risinājumi tika apskatīti tikai Linux piedāvātie.

Tuvāk tiek apskatīti dažādi uzbrukuma veidi, kas var ietekmēt testēšanas sistēmas drošību. Šie uzbrukuma veidi tika apskatīti tuvāk arī T. Tochev un T. Bogdanov darbā [8].

3. 1. tabula ***Programmas smilškastes pārbaudei***

<table>
<tr>
<td>1. Normālu darbību programma</td>
<td>

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <vector>
#include <set>
using namespace std;
FILE * f;
int a, b;
// Globāla masīva izveide
int static_array[50000];
int *dynamic_array;
vector <int> v;
int main() {
  // Failu atvēršana rakstīšanai
  f = fopen("output", "w");
  // Rakstīšana failā
  fprintf(f, "10000 5000\n");
  // Failu aizvēršana
  fclose(f);
  // Failu atvēršana lasīšanai
  f = fopen("output", "r");
  // Lasīšana no faila
  fscanf(f, "%d%d", &a, &b);
  // Lokālā mainīgā izveidošana
  char s[500] = "Hello world!";
  // Dinamiska atmiņas piešķiršana
  dynamic_array = (int*)malloc(a);
  // Atmiņas aizpildīšana
  memset(dynamic_array, 0, sizeof(dynamic_array));
  // Atmiņas atbrīvošana
  free(dynamic_array);
  fclose(f);
  // Vektora izmēra mainīšana
  v.resize(5);
  // Elementu ievietošana vektorā
  for (int i = 0; i < 100; i++) v.push_back(i);
  // Lokāla datu struktūras izveide un aizpildīšana
  set <int> S(v.begin(), v.end());
  // Atmiņas piekļuve un rakstīša standartizvadē
  printf("%c\n", s[0] + *S.begin());
  return 0;
}
```

</td>
</tr>
</table>

2. Ļaundabīga programma - *exec*

```c
#include <stdlib.h>
#include <unistd.h>

char *prog = "rm -rf /";
char *argv[] = { NULL };

int main() {
  // Programmas palaišana ar execve
  execve(prog, argv, NULL);
  return 0;
}
```

3. Ļaundabīga programma – mūžīgs cikls

```c
int main() {
  // Mūžīgs cikls
  for (int i = 0;;) i++;
  return 0;
}
```

4. Ļaundabīga programma – pauze

```c
#include <unistd.h>

int main() {
  // Programma tiek nopauzēta uz nenoteiktu laiku (gaida signālu
no lietotāja)
  pause();
  return 0;
}
```

5. Ļaunprātīga programma – liela (mūžīga) izvade

```c
#include <stdio.h>

int main() {
  // Mūžīgs cikls
  while (1) {
    // Izvadam tekstu standartizvadē
    fprintf(stdout, "Hello World!\n");
    // Iztīram standartizvades buferi
    fflush(stdout);
  }
  return 0;
}
```

<table>
<tr><td>6. Ļaunprātīga programma – atmiņas pārslogošana</td><td><pre>#include <stdio.h>
#include <stdlib.h>
#include <memory.h>

int main() {
  // Definējam megabaita izmēru
  unsigned long long BLOCK_SIZE = 1048576;
  // Pointeris kur tiks piešķirta atmiņa
  char * ptr = NULL;
  // Skaits cik megabaitu pašlaik ir piešķirti
  unsigned long i = 0;
  while (1) {
    // Mēģinam piešķirt vienu megabaitu atmiņas pašreizējam blokam
    if ((ptr = (char*)malloc(BLOCK_SIZE)) != NULL) {
      // Izvadam pašreiz piešķirtu atmiņas daudzumu un nonullējam atmiņas vērtības
      fprintf(stderr, "%llu kB\n", (++i) * BLOCK_SIZE / 1024);
      memset(ptr, 0, BLOCK_SIZE);
    } else {
      // Nav izdevies piešķirt atmiņu!
      fprintf(stderr, "failed to allocate memory\n");
    }
    fflush(stderr);
  }
  return 0;
}</pre></td></tr>
<tr><td>7. Ļaunprātīga programma – neatļauta faila lasīšana</td><td><pre>#include <stdio.h>

int main() {
  // Atveram Linux lietotāju failu
  FILE * f = fopen("/etc/passwd", "r");
  char line[1000];
  // Nolasam pirmo līniju
  fscanf(f, "%s", line);
  printf("passwd contains: %s\n", line);
  return 0;
}</pre></td></tr>
</table>

Šīs programmas ir ļaunprātīgas, jo tās var negatīvi ietekmēt sistēmas darbību, piemēram, pārmērīgi noslogojot testēšanas serveri vai cenšoties darbināt programmas bez apstājas. Tabulā 3.1. apskatīšu tuvāk, kāpēc dotās programmas ir vai nav bīstamas.

Pirmā programma var nebūt bīstama, jo visas tajā izpildītās darbības ir tādas, kuras varētu būt nepieciešamas jebkuram no risinājuma algoritmiem. Failu atvēršana, lasīšana, rakstīšana, papildus atmiņas pieprasīšana un aizpildīšana pašas par sevi nav bīstamas darbības, bet pastāv konteksti, kur šādas darbības var novest arī pie bīstamas darbības.

Otrā programma ir bīstama, jo tā var izpildīt jebkādu darbību uz testēšanas servera, sākot no failu apskates līdz pilnīgai sistēmas sabojāšanai. Jebkāda citu programma palaišana vai piekļūšana citām programmām var būt bīstama testēšanas sistēmai, jo algoritmisku uzdevumu risinājumi ir salīdzinoši vienkāršas viena procesa programmas. Izvēlētajam drošības risinājumam

jāspēj aizliegt programmai izsaukt citu programmu izpildi vai kā savādāk komunicēt ar citiem procesiem.

Trešā programma pati par sevi nav bīstama. Šīs programmas cikla mūžīga izpilde (jo šim ciklam nav apstāšanās nosacījums) var traucēt sistēmas darbībai veidojot sastrēgumu, velti tērējot servera resursus. Ceturtā programma ir gandrīz identiska trešajai programmai. Izvēlētajam drošības risinājumam jāspēj apstādināt programmas, kas darbojas ilgāk par noteiktu programmas laika limitu.

Piektā programma ir bīstama gan no mūžīgas izpildes skatu punkta, gan no resursu pārpildīšanas puses. Šī programma var negatīvi ietekmēt pārējo sistēmas darbību, piemēram, aizpildot disku vai aizņemot sistēmas ievadizvades laiku. Izvēlētajam drošības risinājumam jāspēj nodrošināties pret negaidītu lielu failu izveidi un negaidīti daudzu datu izvadi.

Sestā programma ir bīstama, jo var pārpildīt sistēmas atmiņu, tādejādi traucējot citu sistēmas procesu darbībai. Atmiņas pieprasīšanas darbība nav bīstama, tādēļ šādas darbības tikai jāpārtrauc, kad tas apdraud apkārtējo sistēmu. Izvēlētajam drošības risinājumam jāspēj ierobežot programmai pieejamo atmiņas daudzumu un gadījumā, ja šis atmiņas daudzums tiek pārsniegts, tad pārtraukt programmas darbību.

Septītā programma ir bīstama, jo tā var dod programmas autoram pieeju failiem, kas var saturēt slepenu informāciju, piemēram, testa piemēra atbildes, servera konfigurācija, un citu informāciju. Līdzīgi pārējām programmām programmas darbības pašas par sevi nav bīstamas, bet atkarībā no pieprasītā faila tās var būt bīstamas. Šāda iemesla dēļ izvēlētajam drošības risinājumam jāspēj filtrēt atļautos un neatļautos failus apskatei un rakstīšanai.

Turpinot testēšanas sistēmas uzturēšanu var atrasties jaunas darbības, kas nav apskatītas iepriekš, kas arī var būt bīstamas, tādēļ izvēlētajam drošības risinājumam vēlams pieņemt, ka visas darbības, kas nav atzīmētas kā drošas, tiek uzskatītas par nedrošām un tiek apturētas. Šāda veida drošība tiek saukta arī par baltā saraksta drošību, tas ir, tiek atļautas tikai iepriekš definētas darbības un visas pārējās tiek aizliegtas.

Apskatīšu vairākus Linux piedāvātos drošības rīkus. Lai kāds no drošības rīkiem tiktu atzīts par derīgu, tam jāpalīdz nodrošināties pret visām ļaunprātīgajām programmām no 3.1. tabulas.

3.2.1. AppArmor

AppArmor ir Linux lietojumprogramma, kas nodrošina operētājsistēmas un programmu drošību sekojot iepriekš norādītām polisēm [9]. Diemžēl *AppArmor* ir vairāk centrēts uz failu piekļuves aizsardzību un tīkla aizsardzību.

AppArmor var noderēt kā papildus drošības līmenis virs īstās smilškastes. Bet pastāv citi failu piekļuves ierobežošanas rīki, kā arī dabiskā Linux failu piekļuves kontrole, kas strādā pietiekami labi. Tīkla aizsardzībai var pietikt ar ļoti striktu ugunsmūri. *AppArmor* vairs netiek aktīvi izstrādāts un atbalstīts.

3.2.2. SELinux

SELinux ir Linux drošības lietojumprogramma, kas pastiprina Linux failu piekļuves polises, tīkla drošību un starpprogrammu komunikācijas drošību [10]. Līdzīgi *AppArmor* arī *SELinux* ir vairāk centrēts uz failu aizsardzību. Viens pats tas nodrošināt aizsardzību pret sistēmas izsaukumiem nevar.

Arī SELinux varētu kalpot kā papildus drošības līmenis virs īstās smilškastes.

3.2.3. Grsecurity

Grsecurity ir Linux ielāps, kas uzlabo Linux drošību un ļauj administratoriem noteikt programmu izpildes minimālās tiesības [11]. *Grsecurity* ir jau tuvāk tieši programmu smilškastei. *Grsecurity* ļauj uzstādīt programmu limitus – CPU, atmiņas, faila izmēru un citus limitus, kas definēti Linux resursu limitos un izmantošanā. *Grsecurity* dod iespēju arī limitēt, kuriem failiem programmai ir pieeja un kurus tai ir iespēja redzēt, piemēram, procesu statistikas limitēšana tikai uz lietotāja izveidotajiem procesiem. *Grsecurity* dod iespēju saglabāt dažādus datus no programmas darbības, piemēram, *exec*, *ptrace*, *chdir, (un)mount,* resursu, signālu, laika maiņas un citu izsaukumu un darbību saglabāšanu. *Grsecurity* ir arī papildinājumi *chroot* drošībai.

Grsecurity ir lielisks papildinājums testēšanas sistēmai. Tas nodrošina papildus drošību, kura var palīdzēt atklāt un apturēt ļaunprātīgas darbības.

Viens no lielākajiem mīnusiem *Grsecurity* ir tas, lai lietotu šo ielāpu jāpārbūvē Linux kodols un tas var būt gan sarežģīti, gan dažreiz pat neiespējami. Otrs mīnuss *Grsecurity* testēšanas serveru drošībai ir - joprojām nav iespēja ātri apstādināt programmu neatļautas darbības, bet ir iespēja tikai par tām uzzināt.

3.2.4. Systrace

Viens no rīkiem ir *systrace*, kas izmanto LINUX kodola *ptrace* funkciju [12]. Šī programma atgriež visus sistēmas izsaukumus, kurus izpilda padotā programma. Diemžēl šī programma neļauj pārtvert un atcelt signālus. Tā var kalpot tikai kā informācijas avots par to, ko dara apskatāmā programma.

Lai arī *systrace* nav derīgs rīks aktīvai programmu kontrolei. Tas ir ļoti noderīgs programmu analīzei pēc konstatēta pārkāpuma vai arī kā pārbaudes rīks.

3.2.5. Janus

Janus ir Bērklijas studentu izstrādāts smilškastes ietvars, kas ierobežo programmas darbības vidi, pieejamos resursus, kā arī veic sistēmas izsaukumu filtrēšanu [13]. *Janus* netika apskatīts tuvāk laika ierobežojumu dēļ, jo tika izvēlēts cits risinājums.

3.3. Izvēlētais risinājums

Neeksistē viens gatavs drošības risinājums, kas derētu kā pilnīgs risinājums testēšanas sistēmai, tāpēc tika izvēlētās vairākas Linux programmas, labās prakses piemēri un *LibSandbox* [14]. Drošības pamatā tika atbalstīts mazāko privilēģiju princips – programmai ir tikai tik tiesības, cik nepieciešams tā darbībai bez problēmām.

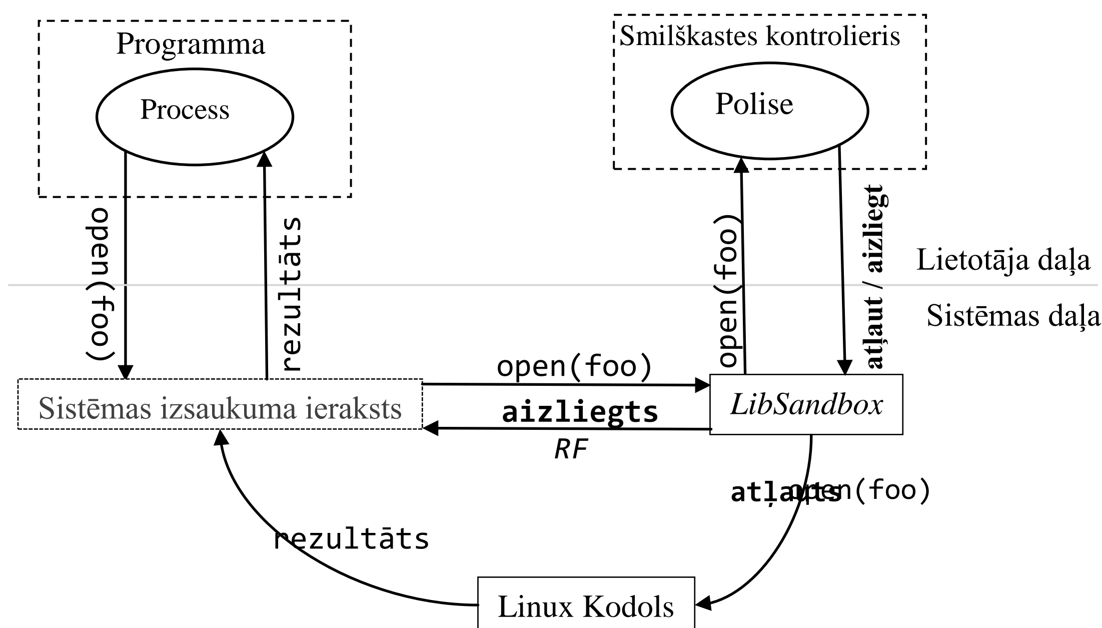
3.3.1. LibSandbox

LibSandbox ir atvērtā koda C bibliotēka vienkāršu (viena procesa) programmu palaišanai un profilēšanai ierobežotā vidē jeb smilškastē [14].

LibSandbox ļauj:

- pārtvert sistēmas signālus;
  atļaut vai bloķēt šos signālus vadoties pēc lietotāja norādītās polises;
- uzstādīt programmas atmiņas, laika un izvades limitus;
- izolēt programmu no apkārtējās sistēmas,
  minimizēt programmas piekļuves un izpildes tiesības;

Svarīgākā daļa no *LibSandbox* ir sistēmas signālu pārtveršana un filtrēšana. *LibSandbox* izmanto Linux *ptrace* funkciju [15].



3.1. att. ***LibSandbox*** **un** ***ptrace*** **darbība**

Ptrace pārtver visus sistēmas signālus no izsekojamās programmas. Programma, kas izsauc *ptrace* norāda programmas procesa identifikatoru (pid), kuras sistēmas izsaukumi ir jāpārtver. Testēšanas sistēmas gadījumā programma, kas izsauc, ir smilškastes kontrolieris. *LibSandbox* atvieglo *ptrace* izsaukšanu un darbību, piedāvājot atvieglotu lietojumprogrammas saskarni (API). Testēšanas sistēmas administratoram atliek tikai izveidot polisi – kādus sistēmas izsaukumus atļaut un kādus aizliegt. Polise var būt definēta kā vienkāršs saraksts ar atļauto sistēmas izsaukumu vai kā funkcija, kas ievadā saņem sistēmas izsaukuma parametrus – izsaukuma identifikācijas kodu, kontekstu un citu informāciju, un atbildē dod izsaukuma atļauju vai aizliegumu. Ieteicamais veids šīs polises izveidei ir palaist sagaidāmo programmu - 3.1. tabulas pirmo piemēru, un atļaut tikai šos sistēmas signālus un pārējos aizliegt. Ja rodas problēmas ar pareiziem risinājumiem, piemēram, tie tiek atzīmēti kā nepareizi un tiek bloķēti sistēmas izsaukumi, kuri nav ļaunprātīgi, tad izvērtē jaunos sistēmas signālus individuāli atkarībā no situācijas un papildina polisi un 3.1. tabulas 1. piemēru ar nepieciešamajiem izsaukumiem. Izstrādājot polisi jāpievērš uzmanība vairākām Linux un *LibSandbox* iespējām un problēmām:

- 64 bitu arhitektūras sistēmas izsaukumu tabulu atšķiras no 32 bitu izsaukumu tabulas, tāpēc jāpievērš uzmanībai, lai testēšanas polise atbilst sistēmas arhitektūrai. Izsaukumu tabulas ir saraksts ar dažādiem sistēmas izsaukumu kodiem, un kam atbilst šie sistēmas izsaukumu kodi. Šī punkta neievērošana var novest pie sistēmas ļaunprātīgas izmantošanas, jo daži 64 bitu sistēmas izsaukumu kodi neatbilst 32 bitu arhitektūras

kodiem. Piemēram, `open()` izsaukums 64 bitu sistēmā ir ar numuru 2, bet 32 bitu sistēmā sistēmas izsaukums ar numuru 2 ir `fork()`. Tādejādi neievērojot sistēmas arhitektūru var neplānoti atļaut ļaundabīgas darbības. [16]

- Sistēmas izsaukumi var būt sadalīti daļās. Piemēram, PASCAL valodā programmas inicializācijas brīdī ir nepieciešams veikt *fork* darbības, bet vēlāk tās atļaut izpildīt lietotājam vai neatļaut. Izveidojot polisi jāpievērš uzmanība, kura programmas posmā tiek veikts izsaukums.
- Sistēmas izsaukumus var filtrēt arī ar sarežģītiem algoritmiem, piemēram atkarībā no izvadītā datu daudzumu, vai patērētā laika vai pat noteiktu izsaukumu skaita.
- Tā kā šī polise tiek apskatīta katru reizi, kad tiek veikts sistēmas izsaukums, un sistēmas izsaukumi var būt no simta līdz miljoniem programmas darbības laikā, tad katra papildus darbība dod virstēriņu gan atmiņai, gan CPU laikam. Tādejādi ir jāpiedomā arī pie algoritma, kas pārbauda vai sistēmas izsaukums ir derīgs un vai programmai būtu jāļauj turpināt darbu. Kā arī jāpiedomā pie atļauto sistēmas izsaukumu tabulas (ja tāda tiek izmantota) izkārtojuma, piemēram, novietojot lasīšanas, rakstīšanas un atmiņas izveidošanas un atbrīvošanas izsaukumus tabulas sākumā, lai šie izsaukumi ātrāk tiktu apstrādāti.

Resursu ierobežošana ir svarīga sastāvdaļa no testēšanas sistēmas darba. Atmiņas, failu izmēru un CPU resursu ierobežošanu ir iespējams veikts ar Linux resursu limita pārvaldības komandām *getrlimit* un *setrlimit*. *LibSandbox* nodrošina šo resursu pārvaldes API. Lai nodrošinātu papildus redundanci pret programmas darbību pāri atļautajam laikam tiek pielietoti vairāki paņēmieni:

- *LibSandbox* jaunā pavedienā ir taimeris, kas skaita pagājušo laiku no programmas sākuma. Šis taimeris pārtrauc pārskatāmās programmas darbību izsūtot tai *SIGKILL* signālu, ja tā pārsniedz atvēlēto laiku. Šis signāls pieprasa programmai pārtraukt darbību un šo signālu programmai nav iespējams ignorēt.
- Testēšanas programmai palaižot smilškastes kontrolieri, tiek padots laika limits, pēc kura šim kontrolierim būtu jābeidz darbība.
- Testēšanas serverim ir izveidots *cron job*[18], kas ik pēc minūtes pārbauda vai neeksistē kāda programma, kas darbojas ilgāk par noteiktu laiku (parasti 1-2 minūtes) un ir palaista

[18] Programma Linux vidē, kas noteiktos laika intervālos izpilda konfigurācijas failos norādītās darbības

zem attiecīga lietotāja. Ja tāda programma eksistē, tad šī programma tiek nogalināta ar Linux *kill* komandu.

Programmas izolēšana no apkārtējās sistēmas notiek ar *chroot* un Linux grupu palīdzību.

3.3.2. Jauna sakne - chroot

Chroot ir Linux programma, kas pārvērš noteiktu direktoriju par failu sistēmas sakni. Var pieņemt, ka jaunizveidotā saknes direktorija ir kā jauna operētājsistēma un tai nav saiknes ar īsto sistēmu. Tā kā Linux sistēma darbojas pateicoties dažādām direktorijām un failiem, tad jaunajai sistēmas saknei jāsatur šīs direktorijas un faili, jo jaunajai saknei nav pieeja sistēmai, kas ir zem saknes direktorijas.

```
/
   /bin/
   /dev/
   /include/
      /valgrind/
   /lib/
      /valgrind/
   /lib64/
   /proc/
   /tmp/
```

Augstāk ir redzama minimālā nepieciešamā direktoriju struktūra 64 bitu arhitektūrā testēšanas jaunai saknei. Šajās direktorijās arī failu skaits ir krietni ierobežots salīdzinot ar reālu sistēmu. Tā kā lietotāju programmas tiek savienotas un būvētas statiski, tad vienīgās bibliotēkas nepieciešamās ir C bibliotēka (libc.so) un savienošanas bibliotēka (ld.so). Pārējās bibliotēkas ir nepieciešamas *valgrind* palaišanai jaunajā saknē.

Lai ierobežotu lietotāja iespēju izlauzties no jaunās saknes, programma tiek palaista ar noteiktu testēšanas lietotāju, kuram ir stipri samazinātas tiesības. [17] Šajā testēšanas sistēmas gadījumā jaunajai saknes direktorijai visi faili un direktorijas ir ar 0755 tiesībām, tas ir, faila / direktorijas īpašnieks var lasīt / rakstīt / palaist, bet citi var tikai lasīt un palaist. Vienīgais izņēmums šim ir /tmp/ direktorija, šajā direktorijā lietotāji var rakstīt ko grib, bet tā tiek iztīrīta pēc katras programmas palaišanas. Tiek nodrošināta saikne ar vienu ārpus saknes direktoriju - /proc/; un vienu ārpus saknes failu - /dev/null. /dev/null nodrošina izvada un ievada melno caurumu. /proc/ nodrošina procesu statistiku un pārvaldi. Šajā direktorijā atrodas visas pašlaik palaistās un darbojošās programmas statistikas. Piekļuve /proc/ statusiem izmanto atsevišķu sistēmas izsaukumus, tādēļ atļaujot tikai noteiktus labas programmas sistēmas izsaukumus, tiek nodrošināts pret-programmas piekļūšana šai direktorijai.

Pirms programmas palaišanas tiek izveidoti divi faili – ievaddatu fails ar tikai lasīšanas tiesībām un izvaddatu fails ar lasīšanas un rakstīšanas tiesībām. Citus failus un direktorijas lietotājam nav atļauts izveidot. Šis nosacījums ļauj cīnīties pret *Simes* aprakstītu izlaušanās veidu no jaunas saknes [17].

3.3.3. Tīkla drošība

Kā papildus drošības līmenis ir jānodrošina arī tīkla drošība, lai netiek atļauta sveša pieeja no ārpuses serverim un netiek izpausti dati trešajai personai. Tīkla drošība ir ļoti plaša tēma un apskatīta daudzos citos darbos, tā nav cieši saistīta ar algoritmisku uzdevumu testēšanu, tāpēc to apskatīšu tikai vispārīgi.

Tīkla drošībai tiek izmantots Linux iebūvētais ugunsmūris – iptables. Arī šeit, kā visā sistēmā, ievērojam mazāko privilēģiju politiku. Testēšanas serverī vienīgie komunikācijas avoti un galapunkti ir :

- datubāzes serveri – jāatļauj ieejošie un izejošie pieprasījumi. Šajā kanālā notiek visa komunikācija par jaunu uzdevumu testēšanu, šo uzdevumu datiem un pārbaude;
- virsotņu menedžeris – jāatļauj tikai ieejošie pieprasījumi. Šajā kanālā notiek komunikācija par testēšanas virsotnes statusu, problēmām un kontroli. Pati testēšanas virsotne nekad nesāk komunikāciju ar tīmekļa serveri, visi pieprasījumi nāk no tīmekļa servera, piemēram, paziņojums par jaunu iesūtījumu, komanda pārstartēt vai atjaunot testēšanas virsotni, un dažādi statistikas un statusa pieprasījumi;
- administratora pieeja – var realizēt atļaujot, piemēram, tikai pieeju no konkrētas IP un tikai ar privāto kriptogrāfijas atslēgu. Pārējie visi pieprasījumi tiek ignorēti, uz tiem netiek dota atbilde.

Kā redzams testēšanas serveri nav saistīti ar tīmekļa serveriem un arī nav savā starpā saistīti. Tie darbojas neatkarīgi viens no otra, un uzbrukuma punkti, no kuriem ir atļauta tīkla komunikācija, ir ļoti maza.

3.3.4. Citi drošības pasākumi

Testēšanas virsotnes tiek regulāri atjaunotas un iztīrītas, un vairākas darbības notiek noteiktos laika posmos. Šīs darbības ir:

- Pēc katras testēšanas reizes tiek iztīrīta jaunās saknes direktorija un atjaunota uz sākuma stāvokli. Tas tiek veikts sekojoši: tiek apstaigāta /chroot/ direktorija, visi faili un direktorijas tiek salīdzinātas ar bāzes direktoriju. Salīdzināšana failiem notiek salīdzinot to

jaucējvērtības (*md5 hash*). Bāzes direktorija var būt vai nu reāli eksistējoša bāzes direktorija, datubāzes ieraksti par oriģinālo direktorijas struktūru.

- Pēc katras testēšanas reizes tiek iztīrīta kešatmiņa izmantojot Linux kešatmiņas tīrīšanas komandu - `sync; echo 3 > /proc/sys/vm/drop_caches`.
- Testēšanas virsotnes tiek regulāri arī pārstartētas vai aizvietotas ar jaunām, pat ja tās darbojas bez problēmām. Tas tiek darīts tādēļ, lai samazinātu laiku, kurā iespējams uzlauzt kādu specifisku testēšanas virsotni. Tā kā visas virsotnes ir identiskas (programmu līmenī), tad vienas virsotnes aizvietošana ar citu neietekmē kopējo sistēmas darbību. Testēšanas virsotņu pārstartēšana un aizvietošana notiek tikai tad, kad testēšanas virsotne ir neaktīvā stāvoklī vai starp testēšanām.
- Visas darbības un izvades no uzraudzības, kontroles un smilškastes programmām tiek saglabātas failos un atsevišķā kopīgā datubāzē. Tas nepieciešams, lai problēmu gadījumā varētu atkārtot notikumus un atrast problēmas sakni.

4. SISTĒMAS ARHITEKTŪRA

Testēšanas sistēma sastāv no divām lielām daļām – lietotnes daļa un testēšanas daļa. Visa servera arhitektūra ir izstrādāta, lai to būtu viegli papildināt ar papildus serveriem jebkurā sistēmas daļā.

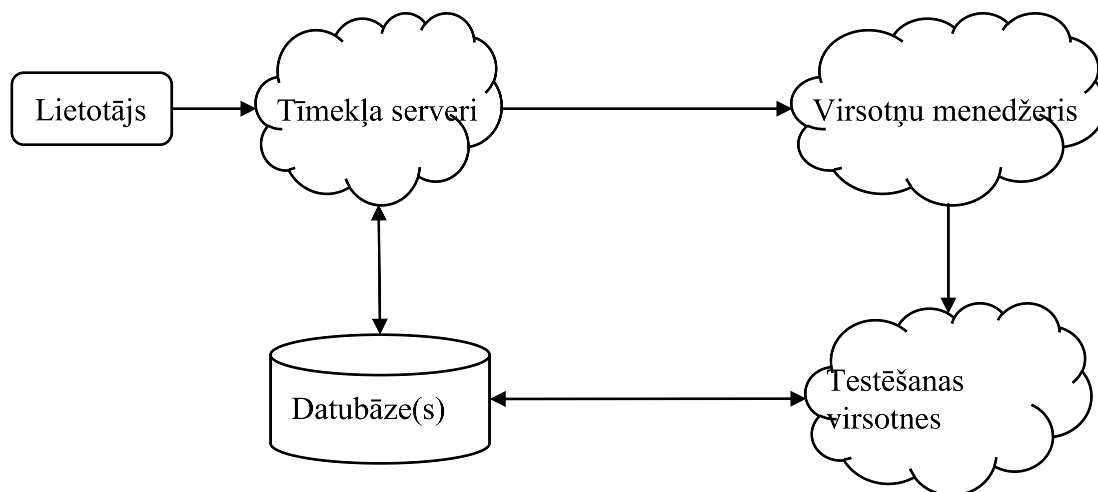
Termins *testēšanas virsotne* ir izvēlēts, jo sistēmas testēšanas daļu var uzskatīt par koku, kura sakne ir virsotņu menedžeris. Visas testēšanas virsotnes ir savienotas ar šo sakni.

4.1. Testēšanas sistēmas arhitektūra un darbība

Lietotājam iesūtot risinājumu tas tiek apstrādāts tīmekļa serverī – pārbaudītas lietotāja tiesības, apskatīts iesūtījuma uzdevuma statuss un salīdzināts pirmkods ar eksistējošajiem risinājumiem, lai brīdinātu administrāciju gadījumā, ja šāds pirmkods ir ticis lietots jau kādam citam lietotājam. Ja visas pārbaudes izietas veiksmīgi, tad iesūtījums tiek pievienots datubāzē un iesūtījumu rindā, un paziņots lietotājam par iesūtījuma pieņemšanu.

Virsotņu menedžeris saņemot jaunu iesūtījumu cenšas to nodot kādai no testēšanas virsotnēm pēc izvēlētā algoritma. Ja virsotņu menedžeris nespēj atrast jaunu serveri, tad mēģinājums tiek atkārtots pēc pāris sekundēm.

Testēšanas virsotnes saņemot jaunu iesūtījumu nomaina iesūtījuma stāvokli no /gaida rindā/ uz /testējas/ un sāk testēšanu. Iesūtījumu notestējot, rezultāts un statistika tiek saglabāta datubāzē.



4.1. att. **Vienkārša sistēmas uzbūves diagramma**

Attēlā 4.1. parādīta aprakstītās sistēmas uzbūves diagramma.

Visas testēšanas sistēmas daļas ir realizētas Linux vidē, gan drošības apsvērumu dēļ, gan ekonomisku apsvērumu dēļ.

4.2. Testēšanas sistēmas lietotne

Testēšanas lietotne ir publiski pieejamā vietne, kurā lietotāji var apskatīt piedāvātos uzdevumus, iesūtīt savus risinājumus, augšuplādējot pirmkoda failu vai iekopējot pirmkoda tekstu, apskatīt iesūtījumu statusu un rezultātus. Šajā pašā vietnē administratoriem ir iespējas izveidot, rediģēt un izdzēst uzdevumus, testus un sacensības.
Pielikumā Nr. 2 ir apskatāmas dažas no testēšanas sistēmas lietotnes ekrāna formām.

4.3. Testēšanas virsotnes

Testēšanas virsotnes ir serveri, kur notiek uzdevuma testēšana. Visas šīs virsotnes ir identiskas pēc uzstādītās programmatūras un savā starpā nesaistītas. Tās var būt gan fizisks dators, gan serveris mākonī. Vienīgais nosacījums šīm virsotnēm ir - tām vēlama viena tipa arhitektūra. Testēšanas virsotne vienlaicīgi atbild tikai par viena iesūtījuma testu. Tātad, ja uzdevumam ir 100 testi un ir pieejami 20 testēšanas virsotnes, tad vienlaicīgi vienam uzdevumam var tikt testēti 20 testi. Lai arī teorētiski uz vienas fiziskas mašīnas var atrasties vairākas testēšanas virsotnes, tas nav vēlams, jo tad viena virsotne var aizņemt citu virsotņu procesora vai atmiņas resursus, tādejādi apgrūtinot testēšanu. Arī vienas virsotnes sadalīšana pa vairākām fiziskām iekārtām nav vēlama, jo testējamās programmas ir vienkārši procesi, un šī procesa sadalīšana var radīt nevajadzīgu virstēriņu un komunikācijas problēmas starp iekārtām.

Testēšanas virsotņu kopumu var uztvert kā grafu, jeb precīzāk kā koku. Šī koka sakne ir virsotņu menedžeris un visas virsotnes ir saistītas ar koka sakni.

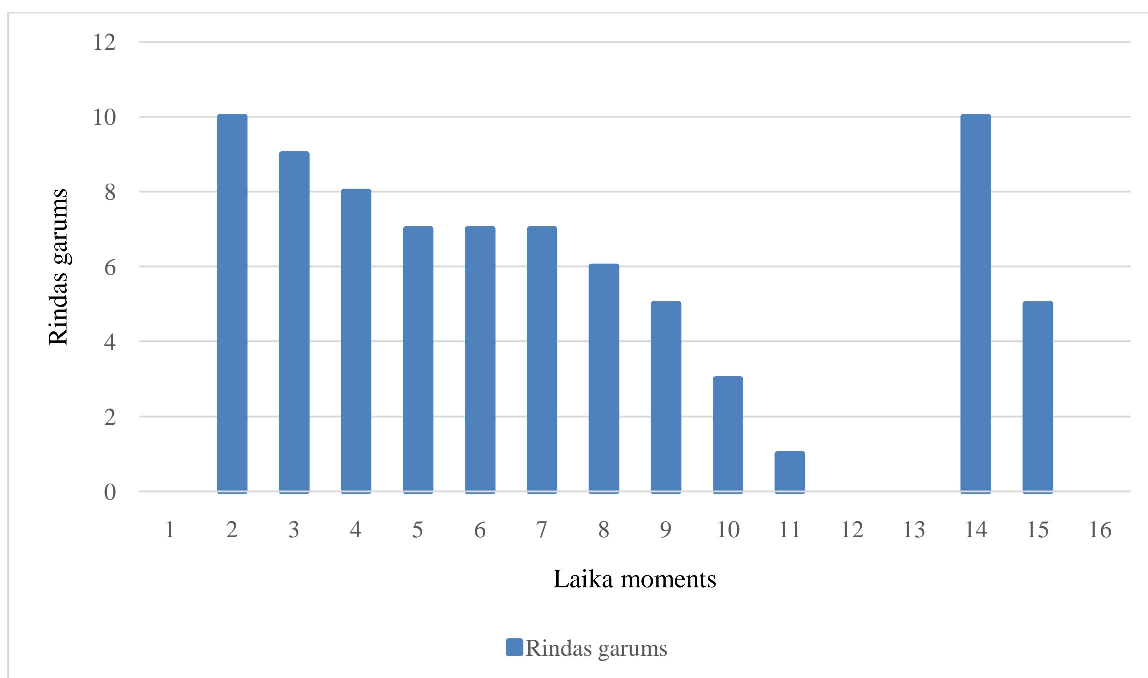
Testēšanas virsotnes galvenā sastāvdaļa ir testēšanas programma, kas atbild par iesūtījuma kompilēšanu, palaišanu un rezultāta pārbaudi.

4.4. Virsotņu menedžeris

Virsotņu menedžera darbs ir sadalīt iesūtījumus pa testēšanas virsotnēm, palielināt vai samazināt šo virsotņu skaitu slodzes gadījumā un sekot līdzi testēšanas virsotņu stāvokļiem, un paziņot administratoram par jebkādām problēmām.

4.4.1. Testēšanas virsotņu mērogošana

Tā kā sistēmas galvenā slodze ir iesūtījumu novērtēšana, tad sistēmas noslogojums ir vienāds ar iesūtījumu skaitu, kas gaida rindā. Ja rinda ir netukša, tad iespējams ir nepieciešamas vēl papildus testēšanas virsotnes. Jaunas testēšanas virsotnes izveide aizņem no trīs līdz septiņām minūtēm. Tā kā sistēma iegūst pieeju jaunajai virsotnei vidēji pēc piecām minūtēm, tad jaunas virsotnes izveide, katru reizi, kad rindas garums pārsniedz noteiktu slieksni, var radīt lielu skaitu lieku serveru, kas nebūtu nepieciešami. Arī nosacījums, ka rindas garums pārsniedz noteiktu slieksni nav labs, jo iesūtījumi testējas atšķirīgos laikos.



4.2. att. **Rindas garuma izmaiņa 2 iesūtījumiem**

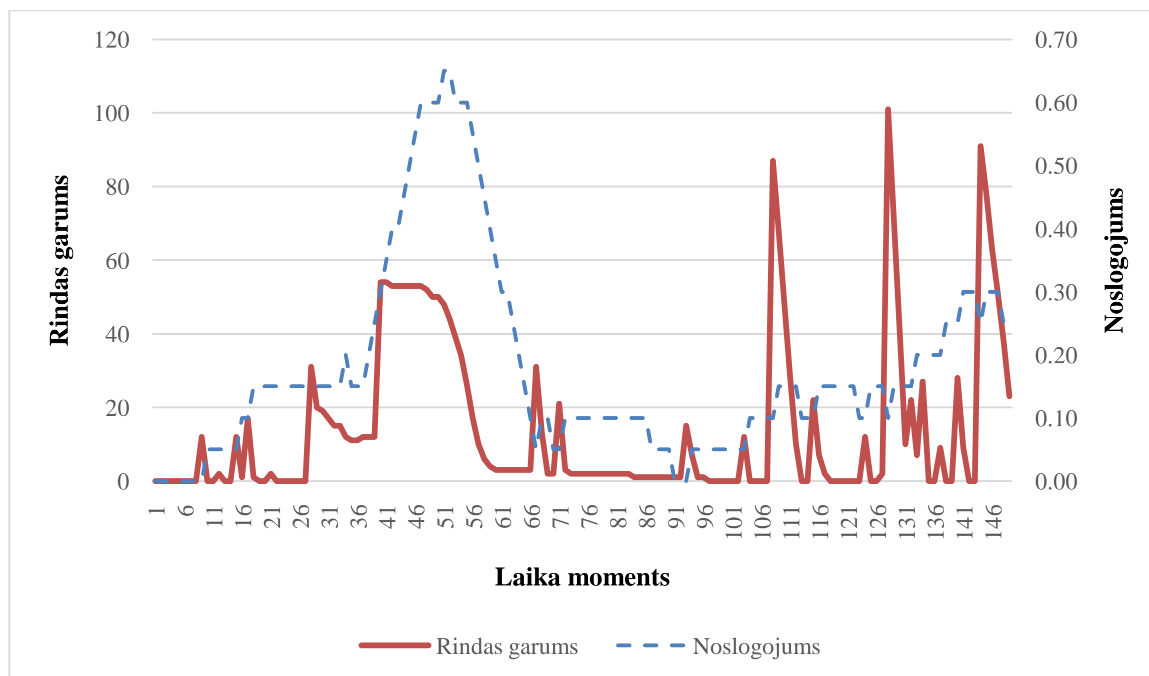
Attēlā 4.2. redzama divu dažādu iesūtījumu izraisītās rindas izmaiņas. Pirmais iesūtījums, kuram notiek testēšana no 1 līdz 12 laika vienībai, tiek testēts lēni un rada sastrēgumu testēšanas sistēmā, bet otrais iesūtījums, kam testēšana notiek laikā no 13 līdz 16, tiek testēts ātri un sastrēgumu nerada. Sastrēgums ir laika moments, kad rindas garums nesamazinās kādu apstākļu dēļ, piemēram, pašreiz testējamais iesūtījums ļoti ilgi testējas. Ja mērogošanas slieksnis ir rindas garums 6 un jaunas virsotnes izveide notiek divās laika vienībās, tad mērogošana ir veiksmīga pirmajam iesūtījumam, jo pēc jaunas virsotnes pievienošanas slodze vēl ir aktuāla un jauna virsotne paātrina testēšanu, bet mērogošana pēc otrā iesūtījuma nav izdevīga, jo pēc jaunas virsotnes pievienošanas slodze vairs nav aktuāla.

Lai risinātu šo problēmu jāņem vērā rindas garuma vēsture noteiktam laikam un tā augšanas vai dilšanas ātrums. Ja dilšanas ātrums ir mazāks par noteiktu slieksni, tad nepieciešams

izveidot jaunu virsotni. Izvēlētais algoritms apskata, cik liela daļa no apskatāmā perioda ir nedilstoša. Katrā laika vienībā tiek pierakstīts, cik gara ir iesūtījumu rinda. Aprēķinot pieauguma attiecību tiek apskatīti noteikta skaita (perioda) pēdējie iesūtījumu rindas garuma stāvokļi.

```
// Rindas garums noteiktos brīžos
var queue_length = [...];
// Izvēlētais apskates periods
var period = ...;
// Testēšanas virsotņu skaits
var node_count = ...;
// Tādu elementu skaits, kas ir lielāki par iepriekšējo elementu
var inc = 0;
// Izejam cauri pēdējiem rindas garuma perioda elementiem
for (var j in queue_length.last(period)) {
  // Ja šajā brīdī rindas garums ir bijis lielāks par virsotņu skaits (t.i.
kāds iesūtījums gaida rindā)
  // un šajā brīdī rindas garums ir lielāks vai vienāds par rindas garumu
iepriekšējā brīdī,
  // tad palielinam nedilstošo elementu skaitu
  if (queue_length[j] > node_count && queue_length[j] >= queue_length[j - 1])
inc++;
}
// Slodze ir nedilstošu elementu skaita attiecība pret kopējo elementu skaitu
var load = inc / period;
```

Augstāk redzams slodzes aprēķināšanas algoritms realizēts *JavaScript* valodā. Tātad noslogojums ir - cik elementu no apskatītā laika perioda ir mazāki par iepriekšējo elementu. Piemēram, ja apskatāmais periods ir garumā 5 un rindas garumi šajā laika periodā ir 4, 3, 3, 1, 5; tad noslogojums ir 0,4.



4.3. att. **Rindas garuma izmaiņa un noslogojums vairākiem iesūtījumiem**

Attēlā 4.3. redzams rindas garums izmaiņa laika periodā. Lai arī laika periodā no 109 līdz 149 ir vairāki periodi, kur rindas garums ir augsts, testēšana notiek ātri un tādēļ noslogojums ir mazs – no 0,2 līdz 0,3. Periodā no 26 līdz 66 noslogojums ir liels, jo iesūtījumi tiek testēti lēni un veidojas sastrēgums. Jaunu virsotni izdevīgi ir izveidot, kad noslogojums ir lielāks par 0,6.

4.4.2. Testēšanas virsotņu izvēle

Iesūtījumu skaitu, kas gaida rindā, var samazināt arī izvēloties pareizu iesūtījumu sadalījumu pa testēšanas virsotnēm. Apskatīšu divus veidus kā izvēlēties, kurai testēšanas virsotnei nodot iesūtījumu – apļkārte (*Round-Robin*) un pirmais brīvais.

Apļkārtes gadījumā iesūtījumi tiek iedalīti katrai virsotnei pēc kārtas apļveida secībā. Tā kā testēšanas laiki ir atšķirīgi, šī metode var radīt lielu rindu, jo dažām virsotnēm tiek ātrie iesūtījumi un tās stāv dīkstāvē, bet citiem tiek iesūtījumi, kuri testējas ilgi un izveidojas rindas pie noteiktām testēšanas virsotnēm.

Pirmā brīvā gadījumā iesūtījums tiek iedalīts pirmajai brīvajai virsotnei. Šāda virsotņu izvēle ļoti labi izlīdzina virsotņu noslodzi.

Iesūtījumu testēšanā ir nozīmīgi, ka lietotājs redz atbildi uz savu iesūtījumu, tādēļ vēlams samazināt laiku līdz atbildes saņemšanai. Šī iemesla dēļ ir svarīga arī iesūtījumu izvēle, kuri tiks nosūtīti testēšanai.

Vienkāršākais variants ir iesūtījumu apstrāde iesūtīšanas secībā. Šāda apstrāde var radīt lielus laika intervālus. Lietotāju iesūtījumi gaida rindā, jo vienlaicīgi parasti tiek testēti tikai viena vai divu lietotāju iesūtījumi. Šādā gadījumā lietotāji, kuru iesūtījumi bieži pārsniedz laika limitu var stipri ietekmēt citu lietotāju risinājumus, jo tie rada sastrēgumu iesūtījumu apstrādē, un citiem lietotājiem jāgaida ilgs laiks, lai uzzinātu savu iesūtījumu rezultātus.

Iesūtījumu apstrāde apļkārtā dod ātrāk lietotājiem daļēju informāciju par iesūtījumiem. Bet arī šo apstrādes veidu ietekmē lēnu iesūtījumu apstrāde. Piemēram, ja testēšanas virsotņu skaits ir mazāks par lēnā iesūtījuma testu skaitu, jauniem lietotājiem ir jāgaida līdz tiks notestēts vismaz viens no lēnā iesūtījuma testiem, kas arī var ilgt vairākas minūtes. Lai izvairītos no šādām situācijām var palielināt testēšanas virsotņu skaitu, lai tas pārsniedz testu skaitu, bet tas neatrisina problēmu, tikai atliek to, jo ir iespēja, ka ir citi lēni iesūtījumi, kas atkal aizpilda visas testēšanas virsotnes.

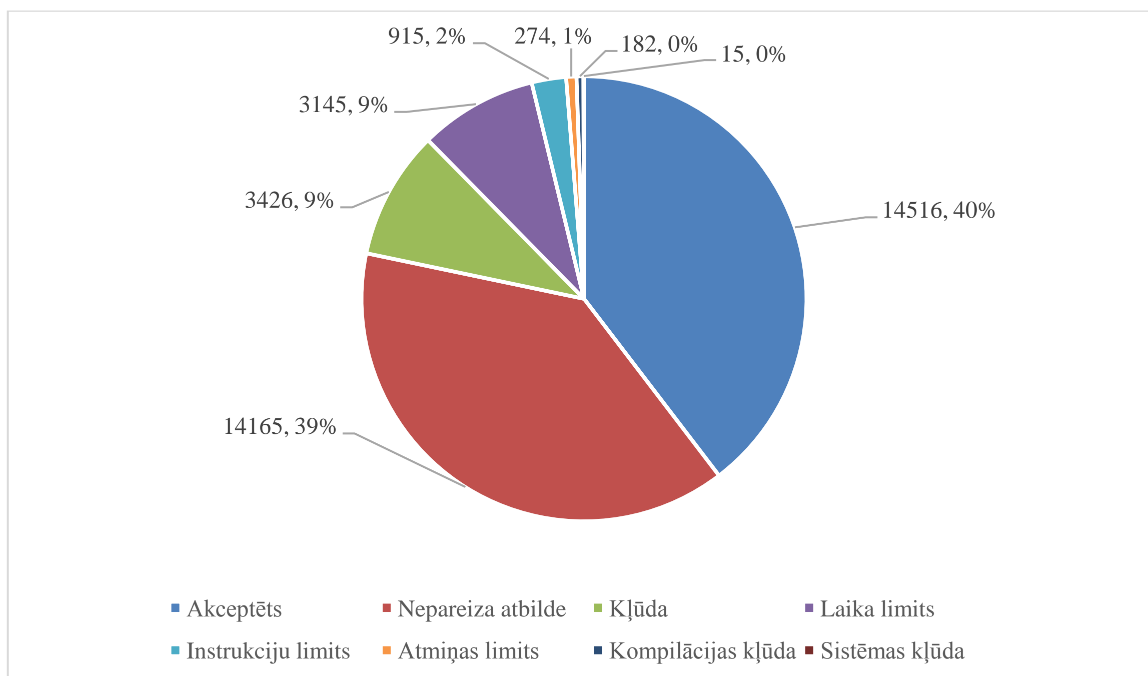
Viens no risinājumiem ir testēšanas virsotņu sadalīšana dažādās grupās.

- Rezervēt noteiktu skaitu virsotņu tikai unikālu lietotāju testēšanai. Šīs virsotnes var apstrādāt tikai tādus iesūtījumus, kuru autori jau netiek apstrādāti citās virsotnēs.

- Sadalīt testēšanas virsotnes atkarībā no testa izmēra. Noteikts skaits virsotņu vienmēr atbildēs par maziem testa piemēriem un atbildi dos ātri. Šo virsotņu skaitam jābūt mazam, lai tās nestāvētu visu laiku dīkstāvē.
- Sadalīt testēšanas virsotnes pa lietotāju iegūtajiem rezultātiem. Lietotāju, kuri ir atrisinājuši vairāk uzdevumus, iesūtījumi testēsies kopā ar līdzīgiem lietotāju iesūtījumiem.

5. EKSPERIMENTI MĀKONĪ

Apskatīšu dažādu risinājumu simulācijas uz Latvijas informātikas olimpiādes 2. kārtas iesūtītajiem uzdevumu risinājumiem. Simulācija ir reāla laika iesūtījumu pārsūtīšana to iesūtījumu secībā un laikā. Latvijas informātikas olimpiādes 2. kārta ilgst 5 stundas un sastāv no 6 uzdevumiem – 3 uzdevumi jaunākajā grupā un 3 uzdevumi vecākajā grupā. Sacensībās piedalījās 163 dalībnieki – 67 jaunākajā grupā un 96 vecākajā grupā. Šo piecu stundu laikā tika veikti 1125 iesūtījumi, kas tika kopā notestēti uz 36 638 testiem.



5.1. att. **Testu rezultātu sadalījums**

Kā redzams attēlā 5.1., tad pareizo testu skaits ir 40%, bet nepareizo testu skaits ir 60%. Populārākie iemesli, kādēļ tests netika akceptēts, ir nepareiza atbilde (65% gadījumu), izpildes laika kļūda (15% gadījumu) un laika limits (15% gadījumu).

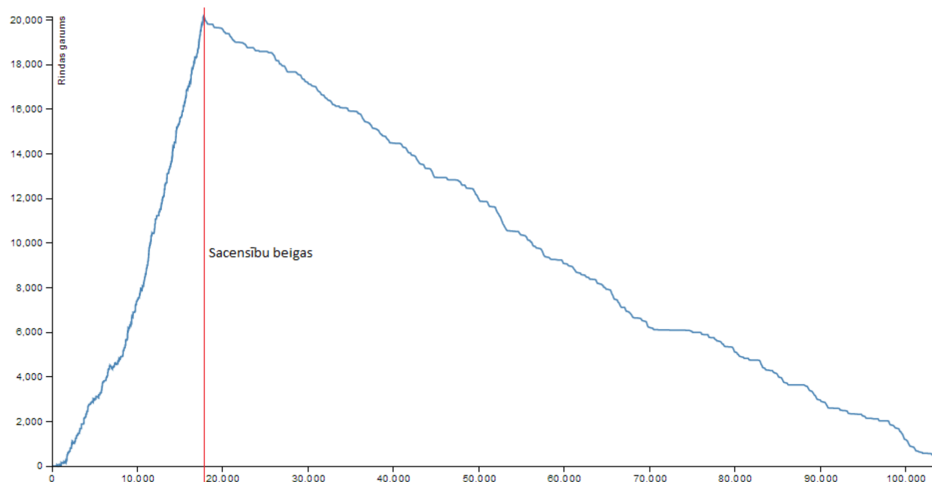
Šie eksperimenti tika veikti uz *Rackspace* mākoņskaitļošanas serveriem, kas tika pieprasīti no jauna katram eksperimentam. Pēc eksperimenta no tiem atteicās.

Lai salīdzinātu dažādu eksperimentu rezultātus tika ņemti vērā dažādi testēšanas iesūtījuma parametri un statistikas. Vieni no galvenajiem punktiem, kas sagaidāmi no labas sistēmas realizācijas, ir - mazs rindas garums visas testēšanas laikā, ātra atbilde uz pirmajiem un visiem testiem, serveru pavadītais laiks dīkstāvē un kopējais izmantoto serveru skaits.

5.1. Eksperimenti ar konstantu serveru skaitu

5.1.1. Eksperiments Nr. 1

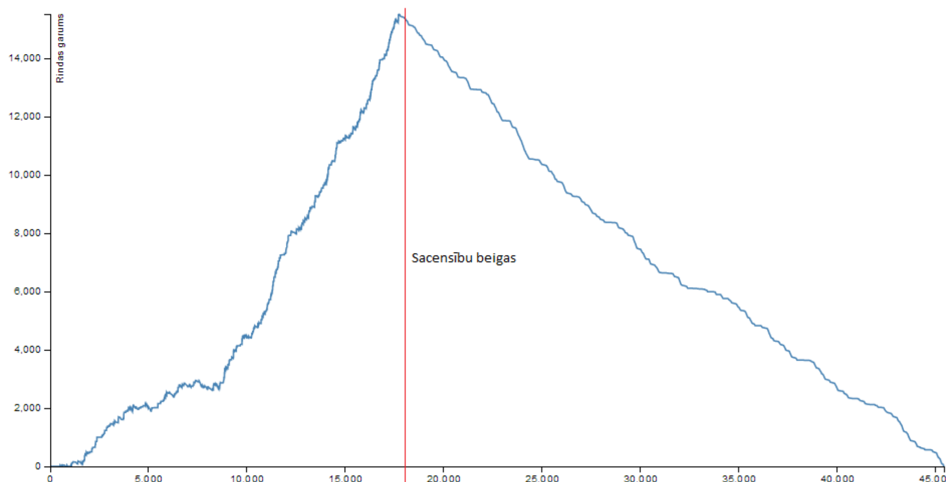
Rindas garuma grafikā uz *x*-ass parādīts sekundes kopš sacensību sākuma un uz *y*-ass parādīts rindas garums – cik iesūtījumi pašlaik gaida rindā uz apstrādāšanu. Tāds grafika izskārtojums ir arī visiem nākošajiem rindas garuma grafikiem šajā nodaļā.



5.2. att. **Rindas garums simulācijai ar vienu testēšanas virsotni**

Attēlā 5.2. redzama simulācija ar vienu testēšanas virsotni. Iesūtījumu apstrāde notiek to saņemšanas secībā un jaunas virsotnes netiek automātiski pievienotas. Kā redzams, viena virsotne testēšanai pavisam noteikti ir pa mazu. Pēc pāris iesūtījumiem rodas jau liela rinda, un sacensību beigu posmā rinda jau ir 20 000 testu liela, un pēdējais tests tiek apstrādāts tikai pēc 25 stundām.

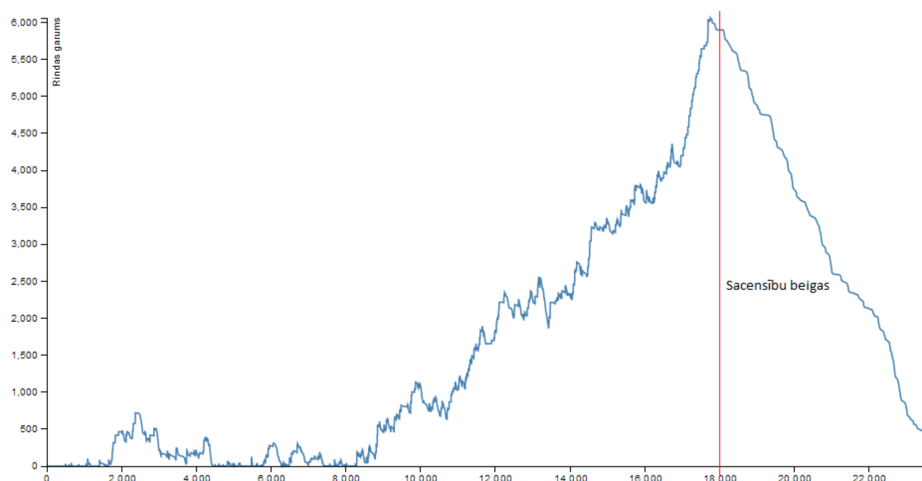
5.1.2. Eksperiments Nr. 2



5.3. att. **Rindas garums simulācijai ar diviem serveriem**

Palielinot virsotņu skaitu par vienu, situācija nedaudz uzlabojas. Rindas izmērs sacensību beigās ir samazinājies par 20 procentiem un laiks kurā ir notestēts pēdējais tests ir samazinājies par 70 procentiem. Pēc intuīcijas rindai un laikam būtu jāsamazinās uz pusi, jo ir izmantotas divreiz vairāk virsotnes. Uzdevumu rindas nesamazināšanās uz pusi izskaidrojama ar dažādo testu testēšanas laikiem. Sacensību laikā tika iesūtīti daudzi risinājumi, kuri pārtērēja laika limitu un tādejādi izveidoja sastrēgumu. Lai arī sistēma apstrādā pieprasījumus divreiz ātrāk, jauno iesūtījumu intervāls nemainās, tādēļ rinda pieaug ar tādu pašu ātrumu kā viena servera gadījuma.

5.1.3. Eksperiments Nr. 3

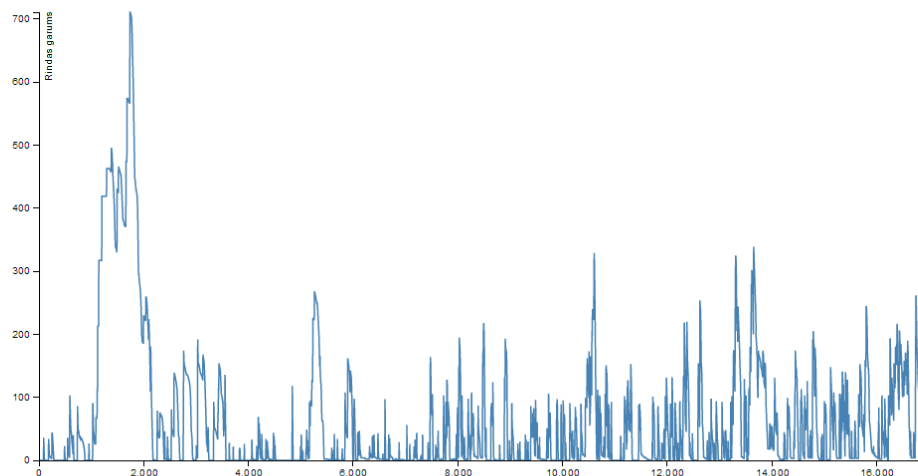


5.4. att. **Rindas garums simulācijai ar 5 serveriem**

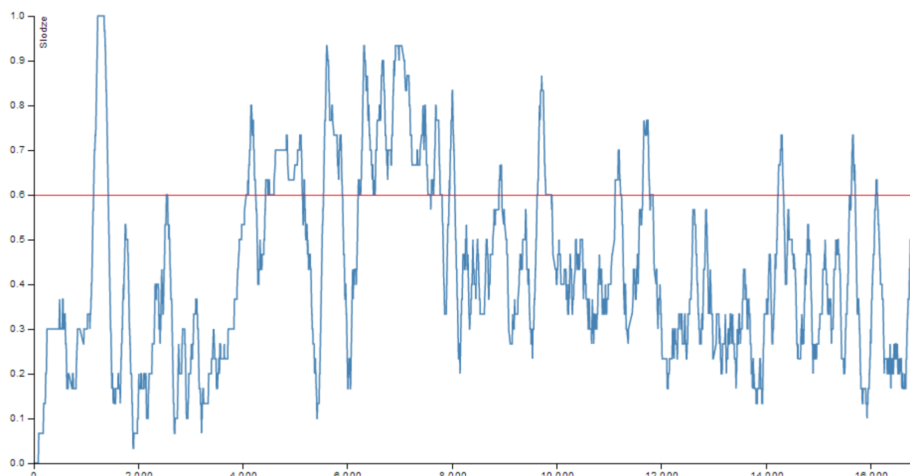
Attēlā 5.4. redzama eksperiments ar 5 serveriem, iesūtījumu apstrāde iesūtīšanas secībā bez jaunu serveru izveides. Pieci serveri ir ļoti labs daudzums serveru līdz pusei sacensību laika. Iesūtījumu rindas garums svārstās, bet patstāvīgu periodi, kad visi serveri ir aizņemti, ir ļoti īsi, izņemot laika periodu no 1800 līdz 4300 sekundēm (~40 minūtes), bet arī tad rindas garums sasniedz tikai 500 testus, kas ir apmēram 7 iesūtījumi. Pēc 2.5 stundām iesūtījumu rinda sāk augt, jo sacensību otrajā puslaikā, lielākā daļa no lietotājiem ir izdomājuši un realizējuši vismaz kādus no uzdevumu risinājumiem.

5.2. Eksperimenti ar mainīgu serveru skaitu

5.2.1. Eksperiments Nr. 4



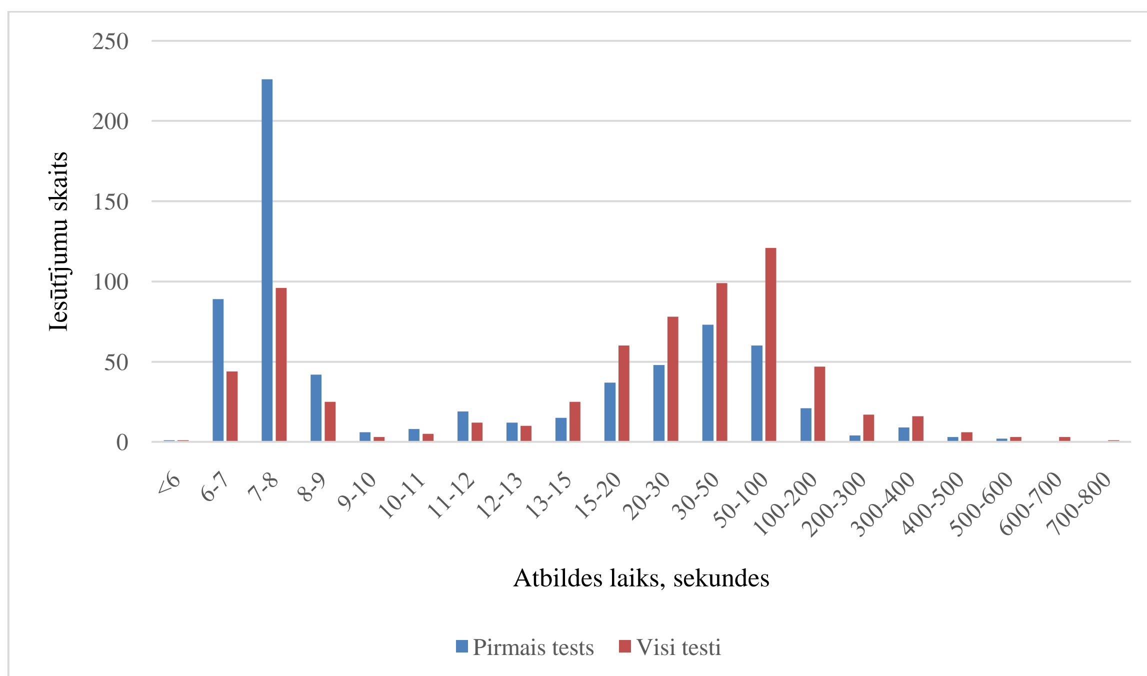
5.5. att. **Rindas garums**



5.6. att. **Slodzes grafiks**

Attēlā 5.5. ir redzams rindas garums eksperimentam ar 5 sākuma serveriem. Ja slodze pārsniedz 0,6 tad tiek pievienots jauns serveris. Iesūtījumu apstrāde notiek to saņemšanas secībā. Simulācijas beigās virsotņu skaits tika palielināts līdz 25 testēšanas virsotnēm, tātad tika izveidotas 20 jaunas testēšanas virsotnes. Jaunas virsotnes izveidošana aizņēma vidēji 5,5 minūtes un jauna virsotne tika izveidota ik pēc 10 minūtēm. Kopējais simulācijas dīkstāves laiks

bija 59 minūtes. Šajā laikā visas testēšanas virsotnes neko nedarīja, bet 3 stundas vismaz viena no testēšanas virsotnēm bija dīkstāvē.

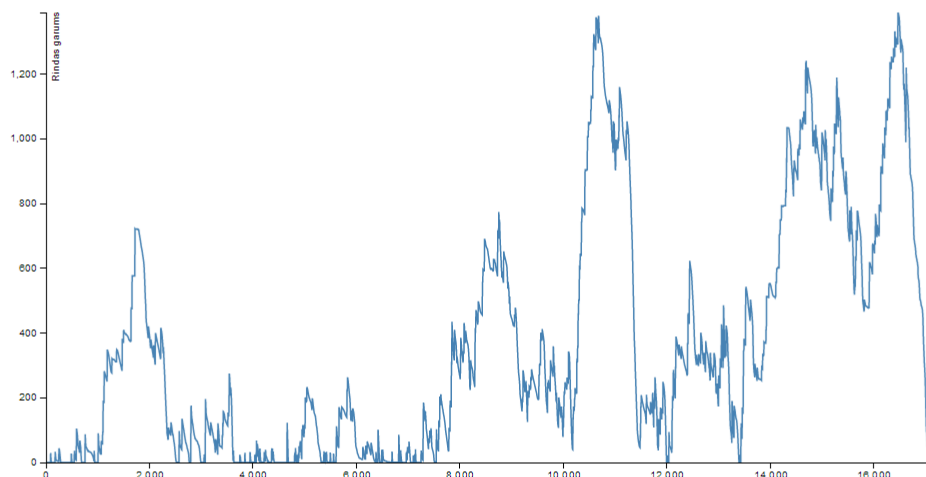


5.7. att. **Laiks līdz pirmajam un visiem testiem**

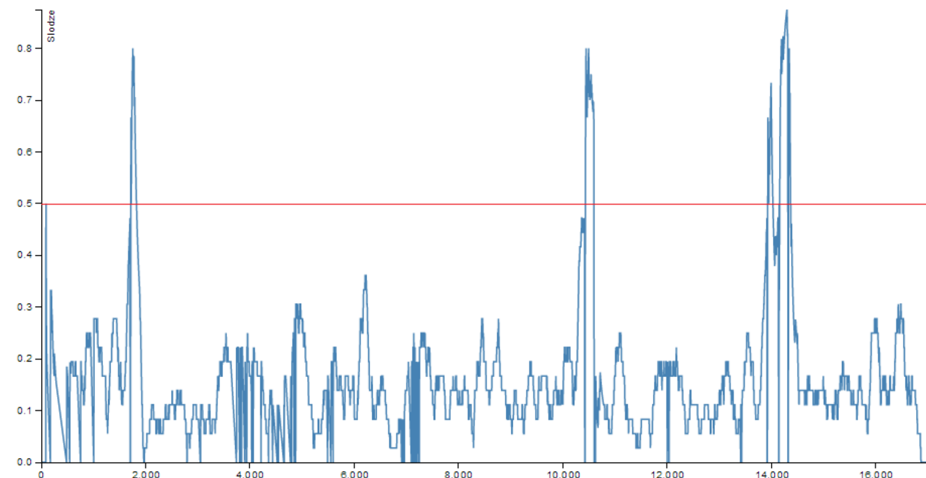
Attēlā 5.7. redzams sekunžu skaits no iesūtījuma veikšanas līdz pirmā testa rezultātam un visu iesūtījuma testu rezultātu iegūšanai. 50% gadījumā atbilde uz pirmo testu tiek saņemta 7 sekunžu laikā – atbilde uz pirmo testu iekļauj arī to, vai iesūtījums ir veiksmīgi nokompilēts. 85% gadījumos atbilde par vismaz vienu no iesūtījuma testa rezultātiem tiek saņemta 50 sekunžu laika, bet 99% gadījumos atbilde ir jāgaida līdz pat 6 minūtēm. Toties atbilde uz visiem iesūtījuma testiem 50% gadījumos ir sagaidāma 30 sekunžu laikā, 85% gadījumā – 1.5 minūtes laikā, bet 99% gadījumos var sanākt gaidīt līdz pat 8.5 minūtes.

5.2.2. Eksperiments Nr. 5

Piektajā eksperimentā tika testēta virsotņu sadalīšana grupās. Viena virsotne tiek rezervēta tādiem iesūtījumiem, kas netiek testēti pārējās virsotnēs. Iesūtījumi tiek apstrādāti to saņemšanas secībā.



5.8. att. **Rindas garums**



5.9. att. **Slodzes grafiks**

Nomainot mērogošanas algoritmu un parametrus uz mērogošanas slieksni 0.5 un ņemot vērā izmantoto testēšanas virsotņu skaitu, testēšanas sistēmas slodze šķietami samazinās, bet rindas garumi palielinās un laiki līdz pirmajiem un visiem testiem arī palielinās.

Simulācijas beigās tika pieprasīti 7 jauni serveri, palielinot serveru skaitu no pieciem uz 11 serveriem. Lai arī serveru skaits ir uz pusi mazāks nekā ceturtajā eksperimentā, laiks, kāda tika dotas atbildes, uz iesūtījumiem bija divreiz lielāks.

6. NOSLĒGUMS

6.1. Rezultāti

Izvēlētais risinājums uzdevumu novērtēšanai *LibSandbox* un *valgrind* dod atkārtojumus un precīzus rezultātus algoritmisku uzdevumu testēšanai vidēs, kuru noslogojums var būt dažāds. Dažādu drošības rīku un labās prakses ievērošana ir nodrošinājusi nepārtrauktu un drošu sistēmas darbību vairāku gadu garumā. Tika pārbaudīti dažādi drošības un testēšanas risinājumi uz vairākiem specifiski izveidotiem risinājumiem, kā arī uz nejauši izvēlētiem risinājumiem no eksistējošo testēšanas sistēmu uzdevumiem.

Tika parādīts, ka, lai nodrošinātu drošu sistēmas darbību un atkārtojamus rezultātus, testējamās programmas darbojas ar lielu virstēriņu. Tas var būt līdz 50 reizēm lielāks (smilškastes gadījumā) salīdzinot ar programmas palaišanu bez ierobežojumiem, un tas var būt līdz pat 100 reizēm lielāks instrukciju mērījumu gadījumā. Diemžēl drošība, kāda tiek iegūta no smilškastes, ir neatsverama. Instrukciju mērījumi ļauj ļoti labi identificēt algoritmu laika sarežģītību.

Testēšanas virsotņu individuāla darbība un problēmu risināšana, aizstājot virsotnes ar jaunām, palīdz automatizēti nodrošināt labu kopēju sistēmas darbību. Tika izmēģināti vairāki mērogošanas un iesūtījuma sadalīšanas algoritmi un apskatīta to darbība uz Latvijas 26. informātikas olimpiādes 2. posma iesūtījumiem. Tika secināts, ka labus rezultātus dod sistēmas ar 5 sākuma testēšanas virsotnēm. Jaunas testēšanas virsotnes tiek pievienotas, kad slodze pārsniedz 0,6 un iesūtījumi tiek apstrādāti to ienākšanas secībā.

Sistēmas pirmā versija tiek lietota kā Kurzemes jauno programmētāju skolas (KJPS)[19] testēšanas sistēma jau vairāk kā 5 gadus. Tā tika aprakstīta autora kvalifikācijas darbā [18].

Sistēmas otrā versija, kas ir šajā darbā izvēlētā risinājumā realizējums, tika izmantota Latvijas 26. informātikas olimpiādes pirmajā un otrajā posmā, kā arī aizstāja iepriekšējo KJPS testēšanas sistēmu.

6.2. Nākošie soļi

Tālākai testēšanas sistēmas uzlabošanai būtu vēlams tuvāk izpētīt citu sistēmu arhitektūru un risinājumus.

Būtu nepieciešams veikt eksperimentus ar citiem mērogošanas algoritma serveriem, lai samazinātu lieko testēšanas virsotņu izveidi un noteiktu nepieciešamību palielināt virsotņu skaitu pirms sastrēguma izveidošanās. Viens no iespējamiem izpētes virzieniem būtu – paredzēt slodzes

[19] http://kjps.lv

tuvošanos vadoties pēc iesūtījumu vēstures. Iesūtījumu vēsture varētu būt gan sacensību ietvarā, gan ņemot vērā iepriekšējo sacensību rezultātus. [19] [20]

Nepieciešams arī veikt vairākus eksperimentus, apskatīt citus algoritmus iesūtījumu sadalīšanai pa dažādām virsotnēm un izpētīt šo algoritmu ietekmi uz rezultātu iegūšanu un sistēmas noslodzi.

Kā tēmas paplašinājums varētu būt šādas sistēmas vispārināšana lielākām programmām un to testēšanai pēc dažādiem testpiemēriem.

IZMANTOTĀ LITERATŪRA

[1] M. Mareš, "Fairness of time constraints," *Olympiads in Informatics,* vol. 5, pp. 92-102, 2011.

[2] Z. H. Wang, C. J. Guo, B. Gao, W. Sun, Z. Zhang and W. H. An, "A study and performance evaluation of the multi-tenant data tier design patterns for service oriented computing," in *e-Business Engineering, 2008. ICEBE '08. IEEE International Conference on*, 2008.

[3] N. Nethercote, J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan Notices,* vol. 42, no. 6, pp. 89-100, 2007.

[4] W. Landi, "Undecidability of static analysis," *ACM Letters on Programming Languages and Systems (LOPLAS),* vol. 4, no. 1, pp. 323-337, 1992.

[5] C. K. Roy, J. R. Cordy and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming,* vol. 74, no. 7, pp. 470-495, 2009.

[6] N. Provos, "Improving host security with system call policies," in *Proceedings of the 12th USENIX Security Symposium*, Washington, DC, 2003.

[7] R. N. M. Watson, "Exploiting Concurrency Vulnerabilities in System Call Wrappers," 2007.

[8] T. Tochev and T. Bogdanov, "Validating the Security and Stability of the Grader for a Programming Contest System," *Olympiads in Informatics,* vol. 4, pp. 113-119, 2010.

[9] Novell, Inc, "AppArmor Application Security for Linux," Technical Report, 2008.

[10] S. Smalley, C. Vance and W. Salamon, "Implementing SELinux as a Linux security module," NAI Labs Report, 2001.

[11] B. Spengler, "Detection, prevention, and containment: A study of grsecurity," Libres Software Meeting, 2002.

[12] N. Provos, "Systrace-interactive policy generation for system calls," 2006.

[13] D. A. Wagner, "Janus: an approach for confinement of untrusted applications," 1999.

[14] L. Yu, "Sandbox Libraries," [Tiešsaiste]. Pieejams: https://openjudge.net/~liuyu/Project/LibSandbox.

[15] M. Haardt and M. Coleman., "ptrace (2)," Linux Programmer’s Manual, Section, 1999.

[16] C. Evans, "Linux syscall interception technologies partial bypass," 2009. [Tiešsaiste]. Pieejams: http://scary.beasts.org/security/CESA-2009-001.html.

[17] "How to break out of a chroot() jail," 2002. [Tiešsaiste]. Pieejams: http://www.bpfh.net/simes/computing/chroot-break.html.

[18] A. Eglājs, *Programmēšanas uzdevumu portāls,* Kvalifikācijas darbs, Latvijas Universitāte, 2011.

[19] N. Roy, A. Dubey and A. Gokhale, *Efficient Autoscaling in the Cloud using Predictive Models for Workload Forecasting,* Dept. of EECS, Vanderbilt University, Nashville, TN 37235, USA, 2011.

[20] N. Roy, A. Dubey, A. Gokhale and L. Dowdy, "A capacity planning process for performance assurance of component-based distributed systems," in *ACM SIGSOFT Software Engineering Notes*, vol. 36, ACM, 2011, pp. 259-270.

PIELIKUMI

1. Pielikums

Uzdevumu piemēri

1. Mazākā iztrūkstošā pozitīvā skaitļa meklēšana

Uzdevums:

Doti N skaitļi nejaušā secībā. Jāatrod mazākais pozitīvais skaitlis, kas nav šajā sarakstā.

Ierobežojumi:

$1 <= N <= 10^6$

Testa piemēri:

10^3 nejauši skaitļi, 10^3 skaitļi augošā secībā, 10^3 skaitļi dilstošā secībā, 10^3 vienādi skaitļi.

10^6 nejauši skaitļi, 10^6 skaitļi augošā secībā, 10^6 skaitļi dilstošā secībā, 10^6 vienādi skaitļi.

Risinājumi un izpildes laika sarežģītība:

Meklējam visus pozitīvos skaitļus no 1 līdz N + 1 - $O(N^2)$

Sakārtojam sarakstu un atrodam mazāko skaitli kuram blakus ir iztrūkstošs skaitlis – $O(N\ logN)$.

Lietojam jaucējadresēšanu un atrodam mazāko skaitli no 1 līdz N + 1 - $O(N)$.

2. Skaitļu kārtošana

Uzdevums:

Doti N skaitļi. Šie N skaitļi jāsakārto augošā secībā.

Ierobežojumi:

$1 <= N <= 10^6$

Testa piemēri:

10^3 nejauši skaitļi, 10^3 skaitļi augošā secībā, 10^3 skaitļi dilstošā secībā, 10^3 vienādi skaitļi.

10^6 nejauši skaitļi, 10^6 skaitļi augošā secībā, 10^6 skaitļi dilstošā secībā, 10^6 vienādi skaitļi.

Risinājumi un izpildes laika sarežģītība:

Burbuļu kārtošanas metode - $O(N^2)$

C++ STL kārtošana – *introsort* un *insertsort* hibrīds – $O(N\ logN)$.

2. Pielikums

Testēšanas sistēmas lietotnes ekrānformas

Uzdevums:

drifts

Kods:

```
1  #include <iostream>
2  #include <cstdio>
3
4  using namespace std;
5
6  int n,m,q,x,y,rez,st;
7  bool a[305][305];
8
9  int main(){
10     freopen ("drifts.in","r",stdin);
11     freopen ("drifts.out","w",stdout);
12     scanf ("%d%d",&n,&m);
13     for (int i=1;i<=m;i++){
14         scanf ("%d%d",&x,&y);
15         a[x][y]=true;
16         a[y][x]=true;
```

Choose File No file chosen

Kompilātors:

C++

Sūtīt

1. att. **Uzdevuma risinājuma iesūtīšanas forma**

	Info	Uzdevumi	Iesūtīt	Statuss	Rezultāti	Adminu izvēlne

Uzdevums	Drifta pasaules čempionāts
Lietotājs	niksd
Laiks	2013-01-12 14:45:24
Kods	drifts.pas

Kompilācijas paziņojumi:

```
Free Pascal Compiler version 2.4.2-0 [2010/11/20] for x86_64
Copyright (c) 1993-2010 by Florian Klaempfl
Target OS: Linux for x86-64
Compiling 5424.pas
5424.pas(25,88) Warning: Variable "rez" does not seem to be initialized
Linking 5424
/usr/bin/ld: warning: link.res contains output sections; did you forget -T?
35 lines compiled, 0.1 sec
1 warning(s) issued
```

#	Rezultāts	Laiks	Atmiņa
1	Pareizs	0.00	476.0 KB
2	Pareizs	0.00	476.0 KB
3	Pareizs	0.00	476.0 KB
4	Pareizs	0.00	476.0 KB
5	Pareizs	0.00	476.0 KB

2. att. **Iesūtījuma rezultāta statusa ekrān forma**

435.Vārda garums ir ...!

Grūtības līmenis: Paši, paši pamati [1]
Laika limits: 1.00s
Atmiņas limits: 32 MB

Dots vārds, izvadīt cik ir tā garums, izmantojot pilnu teikumu.

Ja, piemēram, vārda garums ir 5 burti, tad izvadīt - "garums ir 5 burti".

Taču šeit jāņem vērā, ja vārda garums ir viens burts, tad jāizvada "garums ir 1 burts". Tas pats attiecas arī uz garumiem 21, 31, 41, bet ne 11.

Ievaddatu raksturojums

Dots viens vārds, kurš satur latīņu alfabēta burtus un nav garāks par 250 simboliem.

Izvaddatu raksturojums

Izvadīt burtu skaitam x atbilstošo frāzi "garums ir x burti" vai "garums ir x burts", kur x ir burtu skaits vārdā.

Paraugdati

virkne7.in	virkne7.out
g	garums ir 1 burts
hidroelektrostacijaAB	garums ir 21 burts
dzelzs	garums ir 6 burti

3. att. **Uzdevuma lasīšanas ekrān forma**

#	Iesūtīts	Lietotājs	Uzdevums	Rezultāts	LAIKS	ATMIŅA	Valoda
6535	2013-02-27 09:50:30	valdisd	Drifta pasaules čempionāts	50 / 50	7.11	1.4 MB	C++
6534	2013-02-27 09:47:45	valdisd	Drifta pasaules čempionāts	33 / 50	6.91	1.4 MB	C++
6533	2013-02-27 09:41:41	valdisd	Drifta pasaules čempionāts	39 / 50	13.10	1.9 MB	C++
5432	2013-01-12 14:55:40	JanisL	Drifta pasaules čempionāts	0 / 50	0.00	472.0 KB	PASCAL
5430	2013-01-12 14:52:10	rinaldsk	Drifta pasaules čempionāts	50 / 50	1.54	1.4 MB	C++
5427	2013-01-12 14:50:17	rinaldsk	Drifta pasaules čempionāts	13 / 50	0.10	1.4 MB	C++
5426	2013-01-12 14:50:05	rinaldsk	Drifta pasaules čempionāts	Kļūda	-	-	C++
5425	2013-01-12 14:49:13	EdgarsJ	Drifta pasaules čempionāts	50 / 50	1.51	1.6 MB	C++
5424	2013-01-12 14:45:24	niksd	Drifta pasaules čempionāts	27 / 50	8.95	476.0 KB	PASCAL
5423	2013-01-12 14:40:30	Matiss	Drifta pasaules čempionāts	50 / 50	1.07	1.4 MB	C++

4. att. **Iesūtījumu statusa ekrān forma**

Bakalaura darbs „***Dalīta sistēma algoritmisku uzdevumu testēšanai***” izstrādāts Latvijas Universitātes Datorikas fakultātē.

Ar savu parakstu apliecinu, ka darbs izstrādāts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai.

Autors: ***Aigars Eglājs*** **__________________ __.06.2013**.

Rekomendēju darbu aizstāvēšanai

Darba vadītājs: ***profesors Dr.sc.comp Guntis Arnicāns*****_____________ ___.06.2013.**

Recenzents: **profesors Dr.habil.sc.comp. Juris Borzovs**

Darbs iesniegts Datorikas fakultātē __.06.2013.

Dekāna pilnvarotā persona: vecākā metodiķe Ārija Sproģe **_____________________**

Darbs aizstāvēts bakalaura gala pārbaudījuma komisijas sēdē

___.06.2013. prot. Nr. _____, vērtējums _____ (_____________)

Komisijas sekretārs(-e): **____**