VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS KATEDRA

Magistro baigiamasis darbas

Sinchroninis ir asinchroninis užklausų srautų apdorojimas JAVA programavimo kalbos priemonėmis

Synchronous and asynchronous request stream handling in JAVA

Atliko:

Pavel Blaževič (parašas)

Darbo vadovas:

d m. a. Mindaugas Plukas (parašas)

Recenzentas:

doc. dr. Rimantas Vaicekauskas (parašas)

Vilnius
2006

Turinys

Terminų sutrumpinimų sąrašas 4
Įvadas 5
1. Baziniai konkurentiškumą valdantys modeliai 7
1.1. Gijos pagrindu realizuojamas konkurentiškumo valdymas. 7
1.1.1. Modelio variantai 8
1.1.2. Modelio realizacija 9
1.2. Reagavimo į įvykius pagrindu realizuojamas konkurentiškumo valdymas. 11
1.2.1. Reaktoriaus schema (angl. reactor) 12
1.2.1.1. Reaktoriaus schemos variantai 14
1.2.1.2. Modelio realizacija. 15
1.2.2. Proaktoriaus schema (angl. proactor) 18
1.2.2.1. Proaktoriaus schemos variantai 20
1.2.2.2. Modelio realizacija. 21
2. Bazinių konkurentiškumo valdymo modelių modifikacijos. 23
2.1. Sinchroninis besiblokuojantis užklausų apdorojimas 23
2.1.1. Vienos gijos vienam prisijungimui modelis. 23
2.1.2. Vienos gijos vienam prisijungimui modelis su prisijungimo eile. 25
2.1.3. Vienos gijos vienam prisijungimui modelis su prisijungimo eile ir parametrizuota sukurtu gijų grupe 27
2.2. Sinchroninis nesiblokuojantis užklausų apdorojimas. Reaktoriaus schemos variantai. 30
2.2.1. Vienos gijos ir vieno įvykių išskyrėjo modelis. 30
2.2.2. Atskiros gijos su įvykių išskyrėju ryšio užmezgėjui ir gijos užklausų apdorotojams modelis.. 32
2.2.3. Atskiros gijos ryšio užmezgėjui ir gijos su įvykio išskyrėju užklausų apdorotojams modelis.. 35
2.2.4. Atskiros gijos ryšio užmezgėjui, gijos prisijungimų eilės apdorojimui ir gijos su įvykio išskyrėju užklausų apdorotojams modelis. 38
2.2.5. Atskiros gijos ryšio užmezgėjui, gijos prisijungimo eilės apdorojimui ir parametrizuotas gijų skaičius su savais įvykių išskyrėjais užklausų apdorotojams modelis. 41
2.2.6. Atskiros gijos ryšio užmezgėjui, gijos prisijungimo eilės apdorojimui, gijos su įvykių išskyrėju užklausų apdorotojams bei parametrizuotas gijų skaičius skirtu užklausų vykdytojams modelis. 46
2.2.7. Vienos gijos ir vieno įvykių išskyrėjo modelis su parametrizuota gijų grupe skirta užklausų vykdytojams. 51
2.3. Asinchroninis užklausų apdorojimas (Proaktoriaus schema) 55
2.3.1. Vienos gijos ir grįžtamojo kvietimo modelis. 55
2.3.2. Atskiros gijos ryšio užmezgėjui, gijos prisijungimo eilės apdorojimui ir asinchroniškų užklausos apdorotojų su grįžtamuoju kvietimu modelis. 57
2.3.3. Vienos gijos modelis su parametrizuota gijų grupe pasibaigusių asinchroninių operacijų rezultatų apdorojimui. 60
3. Konkurentiškumą valdančių modelių realizacijų tyrimas. 61
3.1. Tyrimo metodika 61
3.2. Tyrimo vykdymas 63
3.3. Tyrimas 64
3.3.1. Tyrimo aplinka 65
3.3.2. Tyrimo eiga 65
3.4. Pirmos tyrimo dalies rezultatai. 66
3.4.1. Gijos pagrindu realizuojamas valdomas konkurentiškumas. 67
3.4.2. Reaktoriaus schema 68
3.4.3. Proaktoriaus schema 69

3.5. Antros tyrimo dalies rezultatai. 71
3.5.1. Sinchroninis besiblokuojantis užklausų apdorojimas 71
3.5.1.1. Vienos gijos vienam prisijungimui modelis. 71
3.5.1.2. Vienos gijos vienam prisijungimui modelis su prisijungimo eile. 72
3.5.1.3. Vienos gijos vienam prisijungimui modelis su prisijungimo eile ir parametrizuota sukurtu gijų grupe. 73
3.5.2. Sinchroninis nesiblokuojantis užklausų apdorojimas. Reaktoriaus schemos variantai. 75
3.5.2.1. Vienos gijos ir vieno įvykių išskyrėjo modelis. 75
3.5.2.2. Atskiros gijos su įvykių išskyrėju ryšio užmezgėjui ir gijos užklausų apdorotojams modelis. 76
3.5.2.2.1. Modelio b atvejis. 76
3.5.2.2.2. Modelio c atvejis. 78
3.5.2.3. Atskiros gijos ryšio užmezgėjui ir gijos su įvykio išskyrėju užklausų apdorotojams modelis. 80
3.5.2.4. Atskiros gijos ryšio užmezgėjui, gijos prisijungimų eilės apdorojimui ir gijos su įvykio išskyrėju užklausų apdorotojams modelis. 81
3.5.2.5. Atskiros gijos ryšio užmezgėjui, gijos prisijungimo eilės apdorojimui ir parametrizuotas gijų skaičius su savais įvykių išskyrėjais užklausų apdorotojams modelis. 83
3.5.2.6. Atskiros gijos ryšio užmezgėjui, gijos prisijungimo eilės apdorojimui, gijos su įvykių išskyrėju užklausų apdorotojams bei parametrizuotas gijų skaičius skirtu užklausų vykdytojams modelis. 84
3.5.2.7. Vienos gijos ir vieno įvykių išskyrėjo modelis su parametrizuota gijų grupe skirta užklausų vykdytojams. 86
3.5.3. Asinchroninis užklausų apdorojimas (Proaktoriaus schema) 87
3.5.3.1. Vienos gijos ir grįžtamojo kvietimo modelis. 87
3.5.3.2. Atskiros gijos ryšio užmezgėjui, gijos prisijungimo eilės apdorojimui ir asinchroniškų užklausos apdorotojų su grįžtamuoju kvietimu modelis. 89
3.5.4. Išteklių panaudojimas. 91
3.6. Trečios tyrimo dalies rezultatai. 92
3.6.1. Vienos gijos vienam prisijungimui modelis su prisijungimo eile ir parametrizuota sukurtu gijų grupe. 92
3.6.2. Vienos gijos ir vieno įvykių išskyrėjo modelis su parametrizuota gijų grupe skirta užklausų vykdytojams. 93
3.6.3. Vienos gijos ir grįžtamojo kvietimo modelis. 94
Išvados 96
Summary 99
Literatūros sąrašas 100

Terminų sutrumpinimų sąrašas

I/O – Input/Ouput – įvedimo arba išvedimo operacijos

NIO – New I/O, naujas nesiblokuojantis įvedimo ir išvedimo posistemis

Cache – sparčioji atmintis

Socket – tinklinio susijungimo kanalas

Įvadas

Pastaruoju metu nesunku pastebėti didžiules Interneto augimo tendencijas, po kuriomis slepiasi vis didėjantis įvairiausių paslaugų spektras pasiekiamas iš globalinio tinklo. Statiniai tinklalapiai sudaro santykinai labai menka dalį to, kas yra dabar dinamiškai suformuojama ir prieinama kiekvienam tinklo naudotojui, kaip pavyzdžiui nuolatinis vertybinių popierių biržos indeksų sekimas, TV, radijo ar kitų medijų transliavimas, elektroninė komercija, interaktyvus žinučių apsikeitimas, bylų dalinimasis ir daug kitų programinių įrangų veikiančių per tinklą. Todėl nesunku pastebėti, kad dažnai susiduriame su viena iš taikomųjų programų kategorijų paskirstytomis sistemomis.

Daugelio atveju, jeigu kalbame apie paskirstytas sistemas, turime galvoje, kad tarp tų sistemų egzistuoja tam tikras bendras ryšis, paremtas atitinkamų pranešimų siuntinėjimu. Bendrai tie pranešimai vadinami užklausomis. Taigi tarp paskirstytų sistemų atsirandą užklausų srautas. Interaktyvių tinklalapių palaikančias seansus lankymasis yra vienas iš užklausų srautų generavimo pavyzdžių. Toks užklausų srautas yra paverčiamas į didelį rašymo ir skaitymo operacijų kiekį, kas savo ruožtu reikalauja didžiulių skaičiavimo išteklių.

Šiuolaikiniuose paskirstytuose sistemose daug dėmesio yra skiriama našiam ir patikimam konkurentiškumą valdančių komponentų darbo organizavimui, kuris apima taip pat ir užklausų srautų apdorojimą. Pagrindiniai reikalavimai keliami aplikacijoms realizuojančioms minėtus komponentus yra efektingas, rentabilus ir visada pasiekiamas užklausų srautų apdorojimas. Aplikacijos turi būti suprojektuotos tokiu būdu, kad visada galėtų tinkamai sureaguoti ir atlaikyti nenumatytas apkrovas, t.y. kai išteklių poreikis perauga jų pateikimo galimybes, aplikacijos neturi pervertinti savo galimybių dėl išteklių valdymo ir kaip galima geriau užtikrinti esamą našumą, kad to nepajaustu suinteresuotos aplikacijos. Užklausas apdorojančios aplikacijos turi laiku aptikti perkrovimo sąlygas ir bandyti prisitaikyti prie jų, negu sumažinti teikiamų klientams paslaugų kokybės arba atmetinėti užklausas prieš tai informavus vartotojus apie tam tikros paslaugos teikimo sustabdymą.

Užklausų apdorojimas gali vykti dviem būdais: sinchroniškai ir asinchroniškai. Skirtumas tarp minėtų būdų yra toks, kad sinchroninis užklausų apdorojimo procesas užsiblokuoja vykdant tam tikrą operaciją, o asinchroninis atvirkščiai, dirba nesiblokuodamas. Tačiau ir vienas, ir kitas būdas kuo puikiausiai gali naudoti besiblokuojantį arba nesiblokuojantį įvedimo ir išvedimo posistemį.

Minėti užklausų apdorojimo būdai yra kuo puikiai realizuojami įvairiausiuose populiariuose programavimo kalbose, kaip pavyzdžiui C/C++ ar C#. Kadangi JAVA yra palyginus jauna ir besikurianti programavimo kalba, kurioje dar tik atsiranda kai kurie posistemiai egzistuojantys jau

kitose kalbose, sinchroninio nesiblokuojančio, o tuo labiau asinchroninio užklausų apdorojimo realizavimas tapo galimas labai neseniai, atsiradus JAVA 1.4 versijai ir sukūrus IBM kompanijos asinchroninio vykdymo bibliotekai.

Darbo tikslas yra teoriškai ir praktiškai išnagrinėti kaip ir kokius žinomus užklausų srautų aptarnavimo sprendimo būdus galima realizuoti JAVA programavimo kalbos priemonėmis

Darbo uždaviniai:

- Išnagrinėti konkurentiškumą valdančius modelius.
- Išanalizuoti jų realizavimo JAVA programavimo kalbos priemonėmis, bandant sumodeliuoti ir sukurti realiai veikiančius komponentus.
- Atlikti sukurtų konkurentiškumą valdančių komponentų tyrimą.
- Pateikti išvadas bei rekomendacijas kaip galima parametrizuoti užklausų srautų aptarnavimo komponentą užtikrinant optimalų išteklių panaudojimo, pralaidumo bei uždelsimo balansą.

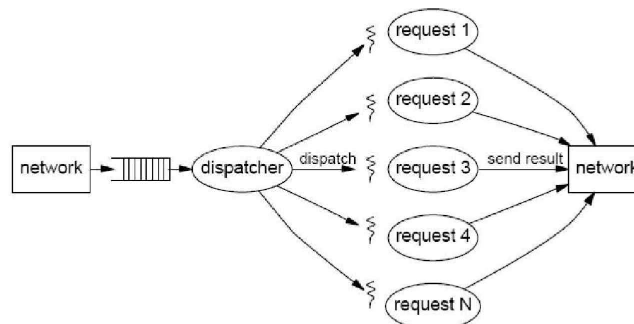
1. Baziniai konkurentiškumą valdantys modeliai

Kaip jau buvo minėta, esminis kiekvienos konkurentiškai veikiančios sistemos suprojektavimas turi atsižvelgti ir palaikyti daug vienu metu ateinančių užklausų srautą. Šiuo metu egzistuoja daug veikiančių sistemų, kuriuos realizuoja vieną ar kitą konkurentiškumo sprendimo būdą. Daugelio atveju tie būdai klasifikuojami į sistemas veikiančias gijų arba reagavimo į įvykius pagrindu.

1.1. Gijos pagrindu realizuojamas konkurentiškumo valdymas.

Gija [WEL02] – skaičiavimo kontekstas, tipiškai susidedantis iš programos skaitliuko, registrų aibės, privataus steko, adresinės erdvės, operacinės sistemos ar programavimo kalbos būsenos susietos su gija. Daugelis gijų gali naudotis ta pačia adresine erdve ir tokiu būdu tiesiogiai bendrauti per bendrai prieinamas duomenų struktūras.

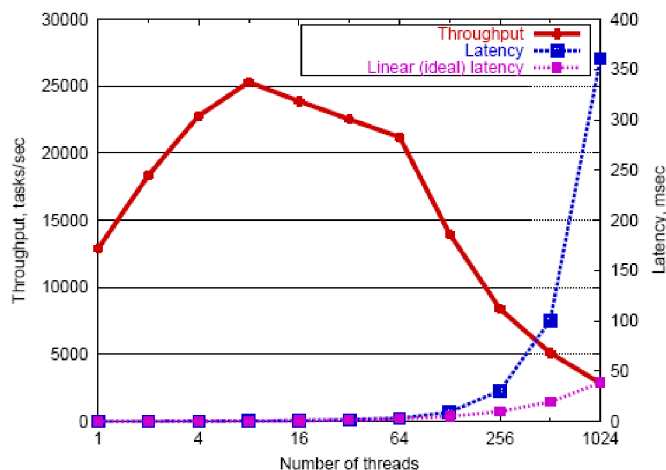
Pats paprasčiausias ir labiausiai intuityvus būdas sukurti konkurentiškumą valdantį serverį tai realizuoti sinchroninį daugiagijį užklausų apdorotoją. „Po giją kiekvienai užklausai" konkurentiškumo modelyje pavaizduotame paveiksle pav. 1, kiekviena priimta užklausa yra deleguojama ir sinchroniškai apdorojama atskiroje gijoje. Kadangi gijos gali prieiti tą pačią adresinę erdvę, sinchronizavimo operacijos, tokios kaip rakinimas ir semaforai, yra naudojamos tam, kad apsaugoti bendrai prieinamus išteklius. Operacinės sistemos kiekvienai veikiančiai gijai užtikrina galimybę naudotis sisteminiais ištekliais skaidraus konteksto keitimo tarp gijų mechanizmo būdu.



pav. 1 Po giją kiekvienai užklausai modelis [WCB01].

Nors minėtas konkurentiškumo modelis yra santykinai lengvai suprogramuojamas, jis kelia tam tikras problemas susijusias su išteklių valdymu ir sistemos plečiamumu.

Sistemos plečiamumas yra glaudžiai susijęs su sistemos našumų, kai žymiai padidėja užklausų apkrovą. Našumas, savo ruožtu, labai priklauso nuo sinchronizacijos, duomenų judėjimo ir konteksto tarp gijų keitimo. Nepalankus efektas gali atsirasti kai gijų skaičius bus daug didesnis nei turimų procesorių skaičius. Tokiu atveju padidės atsakymo į užklausas laikas, nes daugiau laiko bus sunaudojama perėjimui nuo vienos gijos prie kitos, nei naudingam darbui, kurį galėtų atlikti pačios gijos (pav. 2). Geriausiu atveju konteksto keitimo skaičius bus tiesinis, blogiausiu – eksponentinis.



pav. 2 Po giją kiekvienai užklausai modelio efektyvumas [WCB01].

Tam, kad dalinai išspręsti per didelio naudojamų gijų skaičiaus problemą ir su tuo susijusio našumo sumažėjimo efektą, sistemos realizuoja fiksuoto skaičiaus gijų grupę. Šitas sprendimas ateinančias užklausas paskirsto grupėje esančioms laisvoms gijoms, o kai tam tikru momentu laisvų gijų nebebus, ateinančios užklausos bus sustatytos į eilę vėlesniam apdorojimui. Šitas sprendimas nėra 100% teisingas, nes egzistuoja reali galimybė perpildyti operacinės sistemos laukiančių prisijungimų eilę, ko rezultate naujai ateinantys prisijungimai bus automatiškai atmetami be jokio perspėjimo. Kitas sprendimo nepatogumas susijęs su klientų nesąžiningu aptarnavimu, t.y. klientai spėję užmegzti ryši bus aptarnaujami be jokių kliūčių, priešingai kai kiti turės ilgai laukti kol jų prisijungimai bus patvirtinti.

1.1.1. Modelio variantai

§ Vienos gijos vienam prisijungimui modelis.

Kiekvienos užklausos apdorojimas vyksta atskiroje dinamiškai sukurtoje gijoje. Užbaigus apdoroti užklausą gija yra panaikinama.

§ Vienos gijos vienam prisijungimui modelis su parametrizuota sukurtu gijų grupe.

Kiekvienos užklausos apdorojimas vyksta atskiroje paimtoje iš sąrašo gijoje. Užbaigus apdoroti užklausą gija yra grąžinama į sąrašą.

§ Vienos gijos vienam prisijungimui modelis su prisijungimo eile.

Atskiroje gijoje yra užmezgami prisijungimai, kurie vėliau yra įterpiami į prisijungimų eilę. Kitoje gijoje yra laukiama prisijungimų ir startuojama apdoroti kiekvieną užklausą atskiroje dinamiškai sukurtoje gijoje. Užbaigus apdoroti užklausą gija yra panaikinama.

§ Vienos gijos vienam prisijungimui modelis su prisijungimo eile ir parametrizuota sukurtu gijų grupe.

Atskiroje gijoje yra užmezgami prisijungimai, kurie vėliau yra įterpiami į prisijungimų eilę. Kitoje gijoje yra laukiama prisijungimų ir startuojama apdoroti kiekvieną užklausą

atskiroje paimtoje iš sąrašo gijoje. Užbaigus apdoroti užklausą gija yra grąžinama į sąrašą.

1.1.2. Modelio realizacija

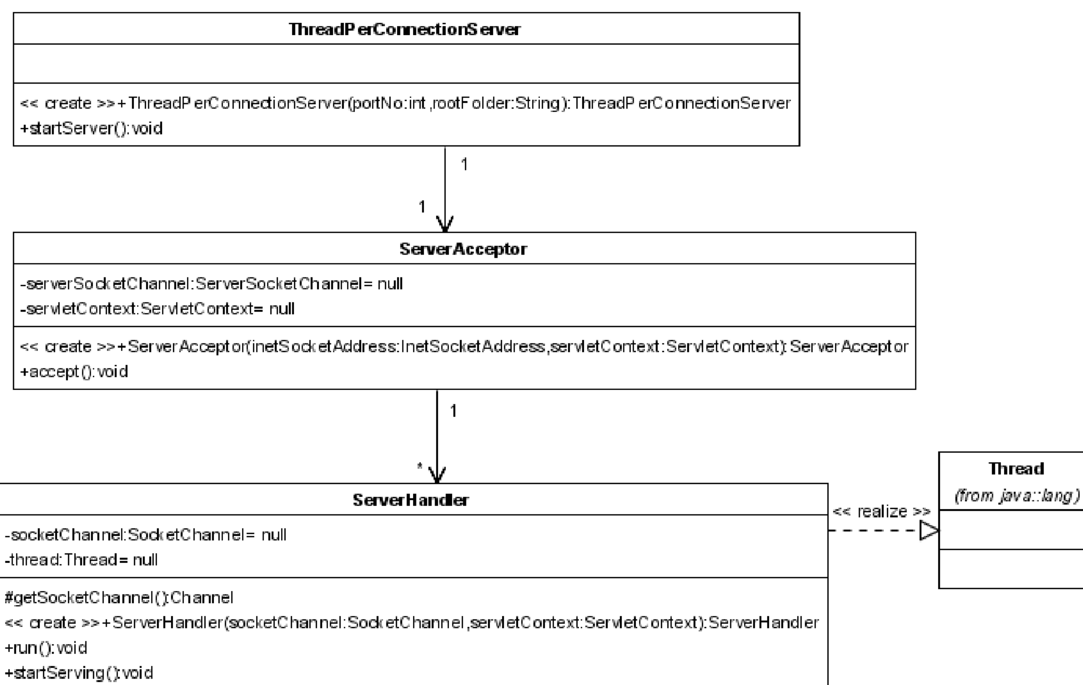
Šio konkurentiškumo modelio realizavimas JAVA programavimo kalbos priemonėmis yra palaikomas nuo ankstyvų JAVA versijų. Iki JAVA 1.4 versijos „vienos gijos klientui" modelis ko gero buvo vienintelis būdas dirbti su konkurentiškomis užklausomis ir apdoroti ryšio užmezgimus.

Kaip jau buvo minėta, šio modelio plečiamumas ir efektyvumas stipriai priklauso nuo naudojamo gijų skaičiaus. Volanomark [NEF03] atlikti testai nurodo, kad tokio modelio veikimas skiriasi priklausomai nuo gijų skaičiaus ir pasirinktos Java platformos. Testų rezultatai nurodomi sekančioje lentelėje lentelė 1.

	Number of Concurrent Connections										
Java Platform	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000	Errors
IBM 1.3.1 Windows	5473	6234	5132								OutOfMemoryError
IBM 1.4.0 Linux											Segmentation fault
Sun 1.3.1 Windows	5263	5482	4282								OutOfMemoryError
Sun 1.4.1 Windows	5457	5997	5899								OutOfMemoryError
BEA 3.1 Windows	5939	5045	4568	4713	3993	4352	3428	3069	3646	3563	
IBM 1.3.1 Linux	4303										OutOfMemoryError
Sun 1.3.1 Linux											OutOfMemoryError
Blackdown 1.4.1 Linux	4306										Hangs (0% CPU)
Sun 1.4.1 Linux	3099										OutOfMemoryError
Sun 1.4.1 Solaris	3452	4513	4364	3877							Segmentation fault
Sun 1.3.1 Solaris	2840	3344	3093	2899							Segmentation fault
BEA 7.0 Windows	3097	1485	1065	787	579	469	349	293	255	224	
BEA 8.1 Windows	1311	1664	1064	776	592	460	346	292	252	223	
Blackdown 1.3.1 Linux	2047	1289	1188	668	789	513	451	283	446	246	
Blackdown 1.3.1 FreeBSD	1574	901	642	556	577	437					Hangs (0% CPU)
BEA 8.1 Linux											Hangs (6% CPU)
Blackdown 1.4.1 FreeBSD											HotSpot VM Error

lentelė 1 VolanoMark tinklinio plečiamumo rezultatai (pranešimų per sekundę) [NEF03]

Realizuojamo JAVA kalboje modelio struktūra yra nurodyta paveiksle pav. 3 .

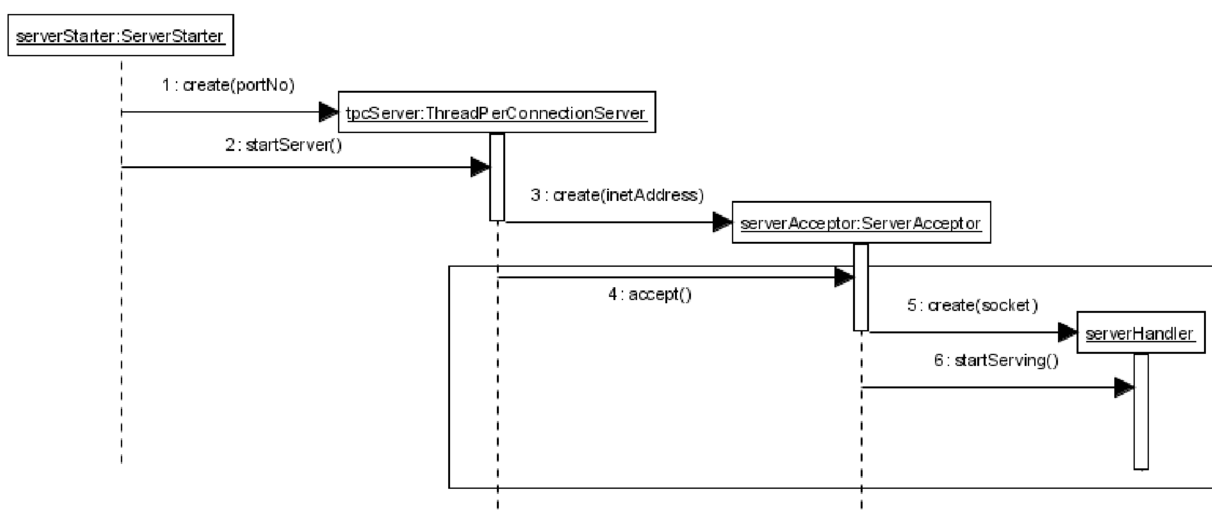


pav. 3 Gijos pagrindu realizuojamo konkurentiškumą valdančio modelio klasės diagrama

Klasių aprašymai:

1. *ThreadPerConnectionServer* – klasė atsakinga už pagrindinį programos ciklą, kuriame yra prašoma ryšio užmezgėjo užmegzti ryšius su klientinėmis aplikacijomis.
2. *ServerAcceptor* – klasė atsakinga už ryšio su klientinėmis aplikacijomis užmezgimą bei paleidimą gijos, kuri apdoros duomenų srautą.
3. *ServerHandler* – klasė atsakinga už sinchronišką duomenų srauto apdorojimą savarankiškoje gijoje.

Modelio veikimas yra parodytas sekos diagramoje pav. 4.



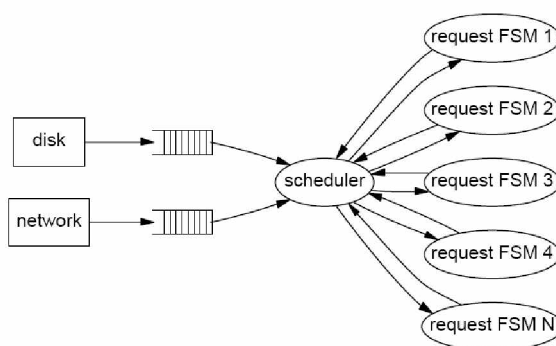
pav. 4 Gijos pagrindu realizuojamo konkurentiškumą valdančio modelio sekos diagrama.

Pavaizduotojoje aukščiau sekos diagramoje galima pamatyti kaip ir kokia tvarka sąveikauja tarpusavyje minėtų klasių objektai.

- Programa prasideda nuo serverio paleidimo.
- Serveris sukuria vienintelį objektą atsakingą už ryšio užmezgimą.
- Cikle serveris prašo ryšio užmezgėjo patvirtinti naują kliento norą užmegzti ryšį. Jeigu jo laukimo eilėje yra laukiančiųjų klientų, ryšis yra užmezgamas ir sukuriamas naujas objektas atsakingas už duomenų apdorojimą. Sukurto objekto operacijos yra įvykdomos sinchroniškai atskiroje gijoje, neblokuojant pagrindinės gijos, kuri galėtų potencialiai vėl užmezginėti naujus ryšius su klientais.
- Pradiniu momentu ir aplikacijos veikimo metu minimalus gijų skaičius yra visada lygus vienam. Apdorojant kiekvieną naują užklausą yra sukuriamos naujos gijos. Sukurtų gijų kiekis nėra programiškai ribojamas todėl maksimalus gijų kiekis gali išaugti iki leidžiamo operacinės sistemos ir JAVA programavimo kalbos galimų gijų ribos.

1.2. Reagavimo į įvykius pagrindu realizuojamas konkurentiškumo valdymas.

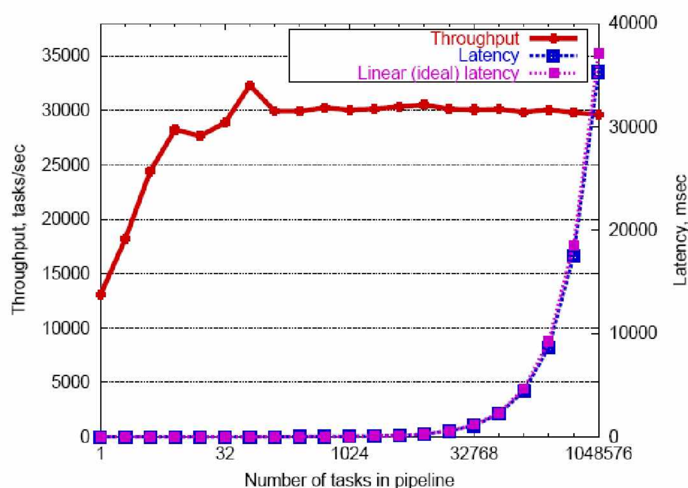
Ankstesnio, gijų pagrindu realizuoto konkurentiškumo modelio plečiamumo apribojimai lėmė reagavimo į įvykius pagrindu realizuojamo konkurentiškumo valdymo principo ir modelio atsiradimą. Kaip nurodytą paveiksle pav. 5, sistema susideda iš mažo gijų kiekio (paprastai po vieną procesoriui), kurios nuolat sukasi apdorodamos įvairaus tipo įvykius paimtus iš eilės. Įvykiai gali būti generuojami operacinės sistemos arba pačios sistemos reikšdami, kad atsirado tam tikro įvykio apdorojimo poreikis, kuris paprastai atitinka prisijungimo, įvedimo, išvedimo ar kitus sistemos specifinius įvykius. Dėka reagavimo į įvykius sprendimo, užklausų apdorotojas yra realizuojamas kaip baigtinės būsenos automatas, kur perėjimai tarp būsenų yra inicijuojami įvykių.



pav. 5 Reagavimo į įvykius pagrindu veikiantis konkurentiškumo modelis [WCB01]

Reagavimo į įvykius pagrindu realizuota sistema užklausų vykdymą atlieka su mažo gijų skaičiaus pagalba, kur „konteksto keitimas“ yra atliekamas kiekvieno užklausų apdorojimo ciklo žingsnio metu. Todėl kiekvienos operacijos apdorojimas turi būti trumpas ir nesiblokuojantis, kai pavyzdžiui įvedimo ar išvedimo į tam tikrą kanalą metu. Iš to seka, kad šis konkurentiškumo modelis turi naudoti nesiblokuojantį įvedimo ir išvedimo posistemį, kuris operuoja dviem etapais:

sistema įvykdo įvedimo arba išvedimo operaciją, kaip tarkim duomenų nuskaitymas iš tinklinio sujungimo kanalo ir iškart grąžina vykdymo kontrolę, nepriklausomai ar I/O užklausa buvo užbaigta ar ne. Jeigu operacija yra užbaigiama vėliau, operacinė sistema generuoja I/O užbaigimo įvykį, kuris gali būti panaudotas tolesniam užklausos apdorojimui. Galima pastebėti, kad įvykių išskyrimas ir apdorojimas tampa labai panašus į gijų tvarkymą. Aprašomas šiame skyriuje konkurentiškumo modelis paprastai yra labiau plečiamas negu gijų pagrindų veikiantis konkurentiškumą valdantis modelis, nes apdorojamos užklausos būsena yra mažesnė negu gijos kontekstas ir susijęs su pastaruoju modeliu dydelis gijų skaičius yra eliminuojamas dėka vienos gijos, kuri pagrinde ir atlieka visą reikiamą darbą. Reagavimo į įvykius pagrindu realizuojamo konkurentiškumą valdančio modelio efektyvumą palyginus su ankstesnio skyriaus modeliu galima pamatyti paveiksle pav. 6.



pav. 6 Reagavimo į įvykius konkurentiškumą valdančio modelio efektyvumas [WCB01].

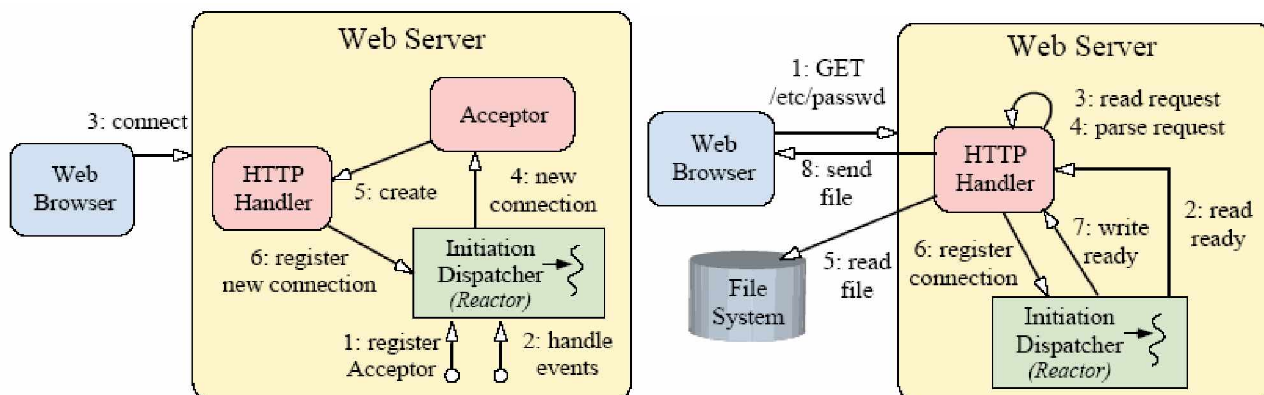
Sekančiuose skyreliuose bus aprašomos tipinės reagavimo į įvykius konkurentiškumą valdančių modelių projektavimo schemos.

1.2.1. Reaktoriaus schema (angl. reactor)

Reaktorius [DOU95] – objektinio projektavimo schema, išskirianti ir informuojanti sinchroninius įvykių apdorotojus, kitaip tariant objektiškai orientuotas būdas atlikti nesiblokuojančias įvedimo/išvedimo operacijas.

Reaktoriaus projektavimo schema apdoroja konkurentiškai ateinančias užklausas. Kiekviena sistemos paslauga gali susidėti iš kelių metodų ir yra reprezentuojama kaip atskiras įvykių apdorotojas, kuris yra atsakingas už paslaugos specifinių užklausų apdorojimą. Įvykių apdorotojų informavimas yra vykdomas su inicijuojančio informatoriaus (kitaip reaktoriaus) pagalba, kuris informuoja užregistruotus įvykių apdorotojus. Paslaugos įvykių išskyrimas yra įvykdomas su sinchroninio įvykių išskyrėjo pagalba.

Kiekvienai paslaugai yra sukuriamas atskiras įvykių apdorotojas, kuris reaguoja į tam tikrų rūšių įvykius. Visi įvykių apdorotojai realizuoja tą pačią sąsają ir yra registruojami inicijuojančiame informatoriuje. Pastarasis komponentas naudoja sinchroninį įvykių išskyrėją, kuris laukią ateinančių įvykių. Atsiradus įvykiams, sinchroninis įvykių išskyrėjas informuoją inicijuojantį informatorių, kuris savo ruožtu sinchroniškai informuoja įvykių apdorotoją asocijuotą su einamuoju įvykiu. Įvykių apdorotojas priima atitinkamą sprendimą ir įvykdo tam tikrą metodą. Šio mechanizmo pavyzdžius galima pamatyti sekančiame paveiksle pav. 9.



pav. 7 Kliento prisijungimo ir HTTP užklausos siuntimas Web serveriui [PHC97+]

Douglas Schmidt‘o sukurtos reaktoriaus projektavimo schemos struktūra susideda iš:

§ Deskriptoriai (angl. *handles*).

Identifikuoja operacinės sistemos naudojamus išteklius. Šiuos išteklius sudaro tinkliniai prisijungimo kanalai, atidarytos bylos, laikmačiai, sinchronizavimo objektai ir t.t. Tokie įvykiai kaip tinklinio prisijungimo, skaitymo ar rašymo yra atstovaujami tinklinio prisijungimo kanalo deskriptoriaus.

§ Sinchroninis įvykių išskyrėjas (angl. *sinchronous event demultiplexer*)

Užsiblokuoja laukdamas įvykių asocijuotų su tam tikra deskriptorių aibe. Vykdymo kontrolė yra grąžinama kai atsiranda galimybė įvykdyti nesiblokuojančią operaciją panaudojant atitinkamą deskriptorių. Įvedimo ir išvedimo įvykių išskyrėjas paprastai yra *select* tipo mechanizmas, kurį galima sutikti UNIX ir Win32 OS platformose. Minėto mechanizmo operacijos įvykdymas nurodo, su kurių deskriptorių pagalba galima atlikti sinchroninę operaciją neblokuojant pagrindinio proceso.

§ Inicijuojantis informatorius (angl. *initiation dispatcher*). Žinomas dar kaip reaktorius.

Apibrėžia įvykių apdorotojų registravimo, pašalinimo ir informavimo sąsajas. Kai sinchroninis įvykių išskyrėjas aptinka naują įvykį, jis informuoja inicijuojantį informatorių, kuris savo ruožtu praneša sistemos specifiniam įvykių apdorotojui. Ryšio užmezgimo, informacijos įvedimo ir išvedimo, duoto laiko pabaigos įvykiai sudaro įprastą įvykių aibę.

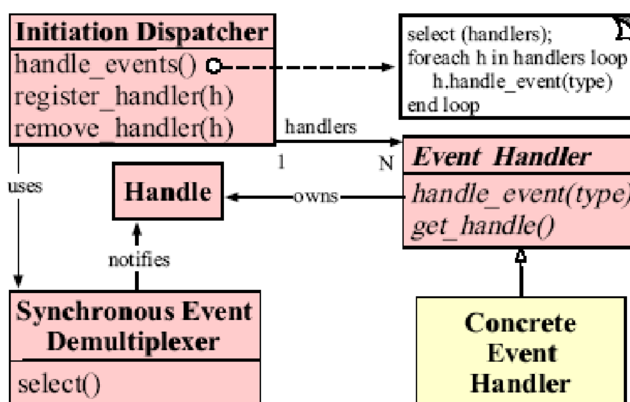
§ Įvykio apdorotojas (angl. *event handler*)

Apibrėžia sąsaja turinčia *hook* metodą, kuris abstrakčiai atstovauja paslaugos specifinę įvykio apdorojimo operaciją. Šitas metodas turi būti realizuotas aplikacijos specifinėje paslaugoje.

§ Konkretus įvykio apdorotojas (concrete *event handler*)

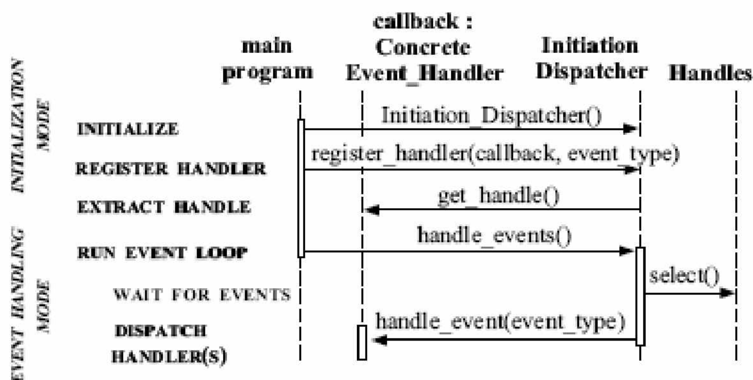
Realizuoja *hook* metodą, o taip pat ir kitus metodus, kurie apdoroja įvykius pagal sistemos logiką. Sistema registruoja konkretų įvykio apdorotoją inicijuojančiame informatoriuje tam, kad jis galėtų apdoroti tam tikros rūšies įvykius. Kai yra sulaukiama šitų įvykių, inicijuojantis informatorius įvykdo atitinkamo konkretaus įvykio apdorotojo *hook* metodą.

Projektavimo schemos klasių diagrama yra nurodyta paveiksle pav. 8.



pav. 8 Reaktoriaus schemos klasių diagrama [DOU95].

Projektavimo schemos sekos diagrama yra nurodyta paveiksle pav. 9.



pav. 9 Reaktoriaus schemos sekos diagrama [DOU95].

1.2.1.1. Reaktoriaus schemos variantai

§ Vienos gijos ir vieno įvykių išskyrėjo modelis.

Visas darbas atliekamas vienoje gijoje panaudojant vienintelį įvykių išskyrėją.

§ Atskirų gijų su įvykių išskyrėjais ryšio užmezgėjui ir užklausų apdorotojams modelis

Viena gija su savo įvykių išskyrėju skirta tik ryšio užmezgimo įvykiams, kita skirta užklausų apdorojimo įvykiams.

- Atskiros gijos ryšio užmezgėjui ir gijos su įvykio išskyrėju užklausų apdorotojams modelis.

  Vienoje gijoje yra užmezgami prisijungimai nenaudojant įvykio išskyrėjo, tuo tarpu kitoje gijoje su įvykių išskyrėju yra apdorojamos užklausos.
- Atskiros gijos ryšio užmezgėjui, gijos prisijungimų eilės apdorojimui ir gijos su įvykio išskyrėju užklausų apdorotojams modelis.

  Vienoje gijoje nenaudojant įvykio išskyrėjo yra užmezgami prisijungimai ir įterpiami į prisijungimų eilę. Kitoje gijoje yra laukiama prisijungimų, kurie vėliau yra registruojami inicijuojančiame informatoriuje veikiančiame trečioje gijoje su savu įvykių išskyrėju.
- Atskiros gijos ryšio užmezgėjui, gijos prisijungimo eilės apdorojimui ir parametrizuotas gijų skaičius su savais įvykių išskyrėjais užklausų apdorotojams modelis.

  Vienoje gijoje nenaudojant įvykio išskyrėjo yra užmezgami prisijungimai ir įterpiami į prisijungimų eilę. Kitoje gijoje yra laukiama prisijungimų, kurie vėliau yra registruojami inicijuojančiame informatoriuje. Minėtas informatorius paskirsto registruojama įvykių apdorotoją vienai iš turimų gijų su savais įvykių išskyrėjais.
- Atskiros gijos ryšio užmezgėjui, gijos prisijungimo eilės apdorojimui, gijos su įvykių išskyrėju užklausų apdorotojams bei parametrizuotas gijų skaičius skirtu užklausų vykdytojams modelis.

  Vienoje gijoje nenaudojant įvykio išskyrėjo yra užmezgami prisijungimai ir įterpiami į prisijungimų eilę. Kitoje gijoje yra laukiama prisijungimų, kurie vėliau yra registruojami inicijuojančiame informatoriuje. Minėtas informatorius naudojantis vienintelį įvykių išskyrėją užklausų vykdymą deleguoja vienai iš turimų gijų.
- Vienos gijos ir vieno įvykių išskyrėjo modelis su parametrizuota gijų grupe skirta užklausų vykdytojams.

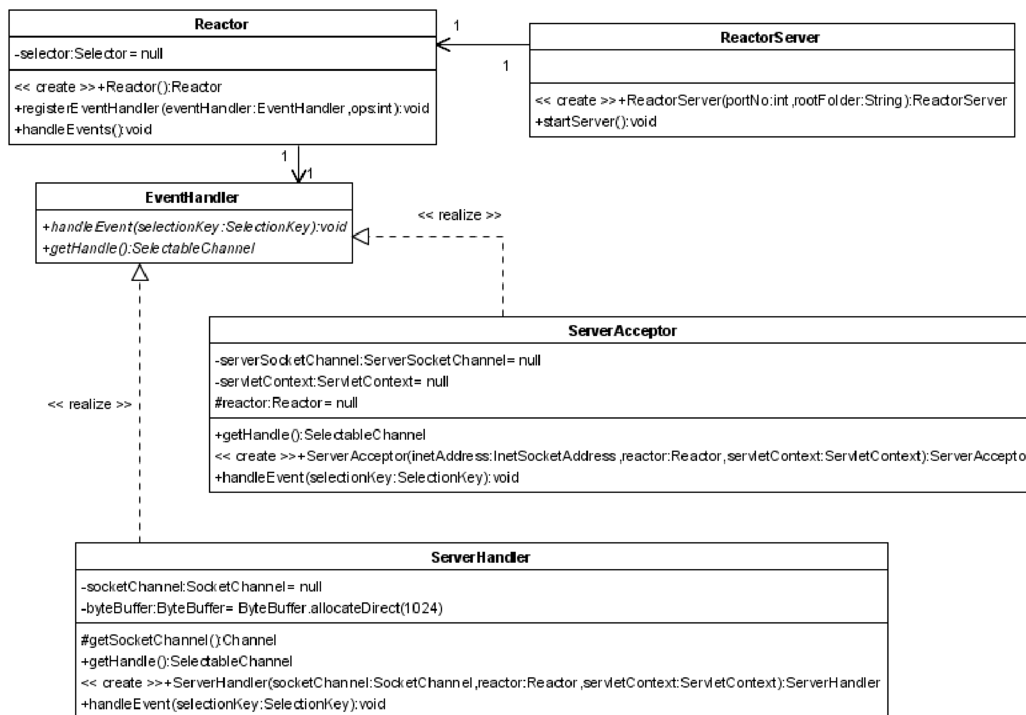
  Panaudojant vienintelį įvykių išskyrėją užklausų vykdymas deleguojamas vienai iš turimų gijų.

1.2.1.2. Modelio realizacija.

Iki JAVA 1.4 versijos praktiškai nebuvo galimybės vienos gijos ribose priiminėti ir apdoroti vienu metu ateinančias konkurentiškas užklausas, o tuo labiau vykdyti nesiblokuojančias įvedimo ir išvedimo operacijas. Tuo metu vienintelis būdas apeiti paminėtus apribojimus buvo naudojamas gijų pagrindu realizuotas konkurentiškumo modelis [DF99] arba *Java Native Interface (JNI)* panaudojimas, kur programuotojai galėjo parašyti kodą kitą programavimo kalba, kaip tarkim C ar C++, kurios jau senai turi nesiblokuojančio rašymo ir skaitymo mechanizmus. Kadangi toks sprendimas yra daugelio atveju priklausomas nuo naudojamos JAVA platformos, plačiai jis

naudojamas nebuvo. Situacija smarkiai pasikeitė su JAVA 1.4 versijos pateikimu, kurioje atsirado nauja įvedimo/išvedimo posistemė *New I/O* (*Nio*). Pagrindiniai šios posistemės privalumai yra tai, kad atsirado galimybė vykdyti nesiblokuojančias įvedimo ir išvedimo operacijas panaudojant tinklinių susijungimų kanalus. Kitas svarbus dalykas tai sinchroninis įvykių išskyrėjas, analogiškas kaip ir C kalboje, kuriuo pagalba ankščiau minėta reaktoriaus projektavimo schema gali būti pilnai realizuota, nenaudojant svetimas JNI realizacijas ir tokiu būdu užtikrindama kodo nepriklausomybę nuo naudojamos platformos.

Realizuojamo JAVA kalboje modelio struktūra yra nurodyta paveiksle pav. 10.

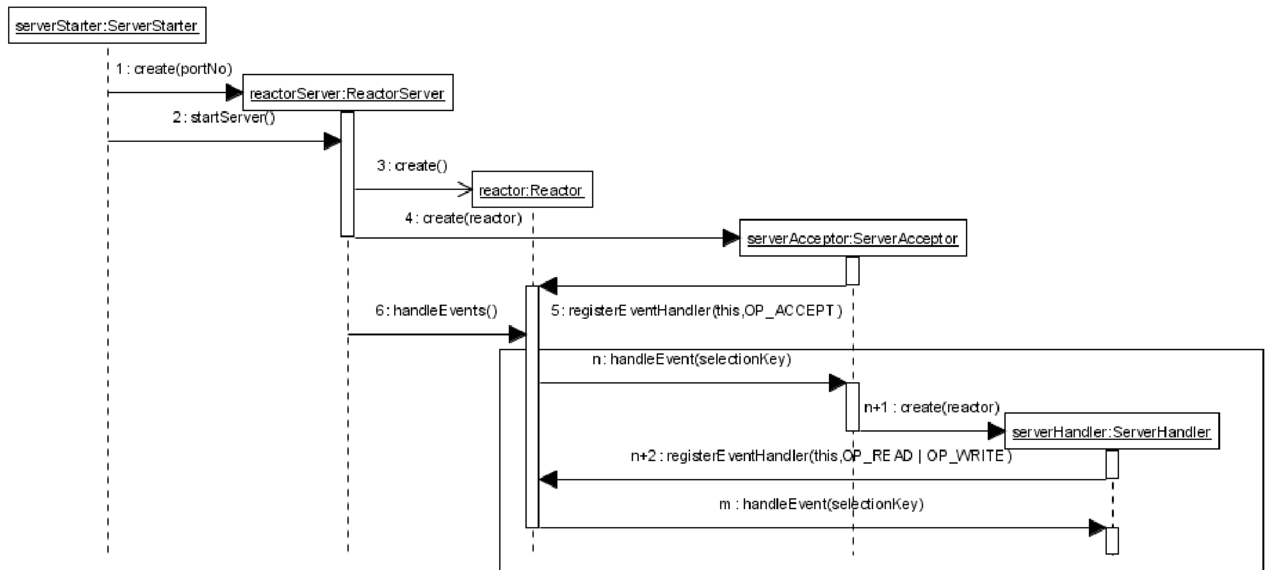


pav. 10 Reaktoriaus schemos pagrindu realizuojamo konkurentiškumą valdančio modelio klasių diagrama.

Klasių aprašymai:

1. *ReactorServer* – klasė atsakinga už reaktoriaus schemos konkurentiškumo mechanizmo paleidimą.
2. *Reactor* – klasė atsakinga už įvykių apdorotojų registravimą, įvykių išskyrimą bei atitinkamų įvykių apdorotojų informavimą
3. *ServerAcceptor* – klasė atsakinga už ryšio užmezgimą ir už įvykių apdorotojų sukūrimą, kurie vėliau yra registruojami reaktoriuje
4. *ServerHandler* – klasė atsakinga už užklausų apdorojimą.

Modelio veikimas yra parodytas sekos diagramoje pav. 11.



pav. 11 Reaktoriaus pagrindu pagrindu realizuojamo konkurentiškumą valdančio modelio sekos diagrama

Pavaizduotojoje aukščiau sekos diagramoje galima pamatyti kaip ir kokia tvarka sąveikauja tarpusavyje minėtų klasių objektai.

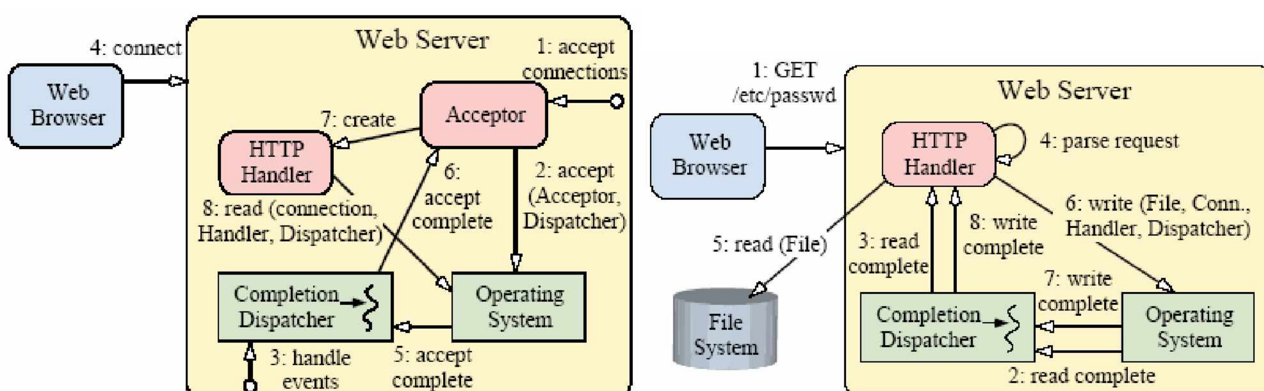
- Programa prasideda nuo reaktoriaus serverio paleidimo.
- Serveris sukuria reaktoriaus objektą.
- Serveris sukuria ryšio užmezgėjo objektą, kuris yra vėliau užregistruojamas reaktoriuje. Registravimo metu yra nurodoma, kad ryšio užmezgėjas yra suinteresuotas tik įvykiais, kurie yra ryšio užmezgimo tipo.
- Serveris paleidžia pagrindinį reaktoriaus ciklą, kurio kiekvienos iteracijos metu sinchroninis įvykių išskyrėjas užsiblokuoja laukdamas įvykių.
- Sinchroninis įvykių išskyrėjas grąžiną sąrašą įvykių, kurie gali būti ryšio užmezgimo, rašymo ar skaitymo operacijų tipo.
- Pagal gautus įvykius reaktorius identifikuoja įvykių apdorotojus bei įvykdo atitinkamas duomenų apdorojimo operacijas. Pvz. kai sinchroninis įvykių išskyrėjas aptinka ryšio užmezgimo įvykį, surandamas yra ryšio užmezgėjo objektas, kuris užmezga prisijungimą, sukuria naują duomenis apdorojantį objektą ir užregistruoja jį reaktoriuje su rašymo ir skaitymo tipo įvykiais. Kai įvykių išskyrėjas aptinka skaitymo ar rašymo įvykius yra identifikuojamas duomenis apdorojantis objektas, kuris vėliau vykdo atitinkamas operacijas.
- Gijų kiekis realizuotame modelyje visada išlieka pastovūs ir lygus vienam.

1.2.2. Proaktoriaus schema (angl. proactor)

Proaktorius [PHC97+] – objektinio projektavimo schema, išskirianti ir informuojanti asinchroninių operacijų užbaigimo įvykių apdorotojus, kitaip tariant objektiškai orientuotas būdas atlikti asinchronines įvedimo ir išvedimo operacijas.

Proaktoriaus projektavimo schema užtikrina įvykių apdorotojų išskyrimą ir informavimą panaudojant asinchroninius užbaigimo įvykius. Šitas konkurentiškumo modelis žymiai palengvina asinchroninių sistemų kūrimą dėka užbaigimo įvykių išskyrėjo ir informatoriaus, kuris užtikrina tinkamą įvykių apdorotojų reagavimą.

Pritaikius proaktoriaus schema, sistema nurodo operaciniai sistemai atlikti tam tikrą asinchroninę operaciją kartu užregistruodama grįžtamojo iškvietimo metodą užbaigimo informatoriuje. Pastarasis savo ruožtu informuoja įvykių apdorotojus operacijos vykdymo užbaigimo atveju. Tada operacinė sistema įvykdo nurodytą operaciją sistemos vardu ir rezultatą padeda į bendrai prieinamą eilę. Užbaigimo informatorius yra atsakingas už užbaigimo pranešimų paėmimą iš minėtos eilės ir atitinkamo užregistruoto grįžtamojo metodo įvykdymą. Šio mechanizmo pavyzdžius galima pamatyti sekančiame paveiksle: pav. 12.



pav. 12 Kliento prisijungimas ir HTTP užklausos siuntimas Web serverio.

Douglas Schmidt‘o sukurtos proaktoriaus projektavimo schemos struktūra susideda iš:

§ Deskriptoriai (angl. *handles*)

Deskriptoriai identifikuoja operacinės sistemos esybes, kurios gali generuoti užbaigimo įvykius, kaip pavyzdžiui tinklo prisijungimai ar atidarytos bylos. Užbaigimo įvykiai yra generuojami atsakant į užklausas, kaip pavyzdžiui į prisijungimo ar duomenų užklausas ateinančias iš išorinių sistemų, o taip pat atsakant į operacijas, kurias generuoja pati sistema, kaip pavyzdžiui pabaigos laiko ar asinchroninio įvedimo ir išvedimo posistemio.

§ Asinchroninės operacijos (angl. *asynchronous operations*)

Asinchroninės operacijos atstovauja potencialiai ilgai trunkančias operacijas, kurios realizuoja tokius dalykus, kaip pavyzdžiui asinchroninį informacijos skaitymą ar rašymą

panaudojant tinklinio prisijungimo kanalo deskriptorių. Po to kai operacija buvo iškviesta, ji yra vykdoma neblokuojant kviečiančios gijos vykdymo.

§ Užbaigimo apdorotojas (angl. *completion handler*)

Užbaigimo apdorotojas apibrėžia sąsaja susidedančią iš vieno ar daugiau *hook* tipo metodų. Šitie metodai sudaro galimų operacijų aibę, kurios gali būti panaudotos apdorojant informaciją atėjusią su sistemos specifiniais užbaigimo įvykiais, kuriuos savo ruožtu generuoja užsibaigusios asinchroninės operacijos.

§ Konkretūs užbaigimo apdorotojai (angl. *concrete completion handlers*)

Konkretūs užbaigimo apdorotojai specializuoja užbaigimo apdorotoją realizuojant tam tikrus paveldėtus *hook* metodus. Šitie metodai apdoroja rezultatus atėjusius su užbaigimo įvykiais, kai asinchroninės operacijos asocijuotos su užbaigimo apdorotoju baigė savo vykdymą.

§ Asinchroninis operacijų procesorius (angl. *asynchronous operation processor*)

Asinchroninės operacijos yra iškviečiamos su tam tikru deskriptoriumi ir įvykdamos iki galo su asinchroninio operacijų procesoriaus pagalba, kuris dažniausiai yra realizuotas operacinės sistemos branduolyje. Kai asinchroninė operacija baigia savo vykdymą asinchroninis operacijų procesorius generuoja atitinkamą užbaigimo įvykį. Jis įterpia sugeneruotą įvykį į užbaigimo įvykių eilę asocijuotą su deskriptoriumi, su kuriuo operacija buvo iškviesta. Šita eilė kaupia užbaigimo įvykius iki to laiko kol jie nebus paskirstyti atitinkamiems užbaigimo apdorotojams.

§ Asinchroninis įvykių išskyrėjas (angl. a*synchronous event demultiplexer*)

Asinchroninis įvykių išskyrėjas – tai funkcija, kuri laukia užbaigimo įvykių užbaigimo įvykių eilėje, kai tie įvykiai bus ten patalpinti užsibaigus asinchroninei operacijai. Asinchroninis įvykių išskyrėjas paima ir pašalina vieną ar daugiau užbaigimo įvykių rezultatus ir grąžina juos kvietėjui.

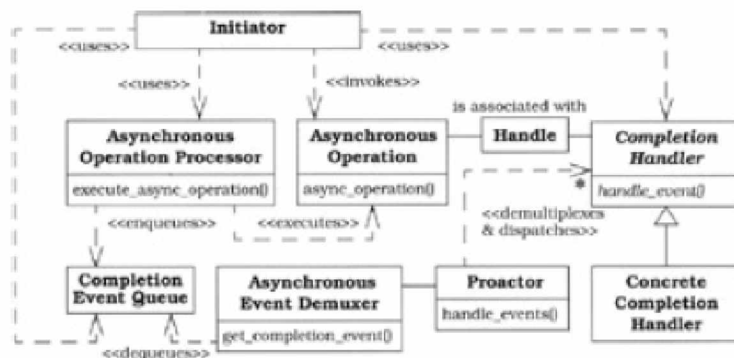
§ Proaktorius (anlg. *proactor)*

Proaktorius suka ciklą, kurio kiekvieno žingsnio metu yra kviečiamas asinchroninis įvykių išskyrėjas tam, kad sulaukti užbaigimo įvykių. Sulaukus vieno ar daugiau įvykių, proaktorius informuoja su įvykių asocijuotą užbaigimo įvykių apdorotoją kviečiant *hook* metodą, kuriame bus apdorojami rezultatai atėję su užbaigimo įvykiu.

§ Iniciatorius (angl. *initiator*)

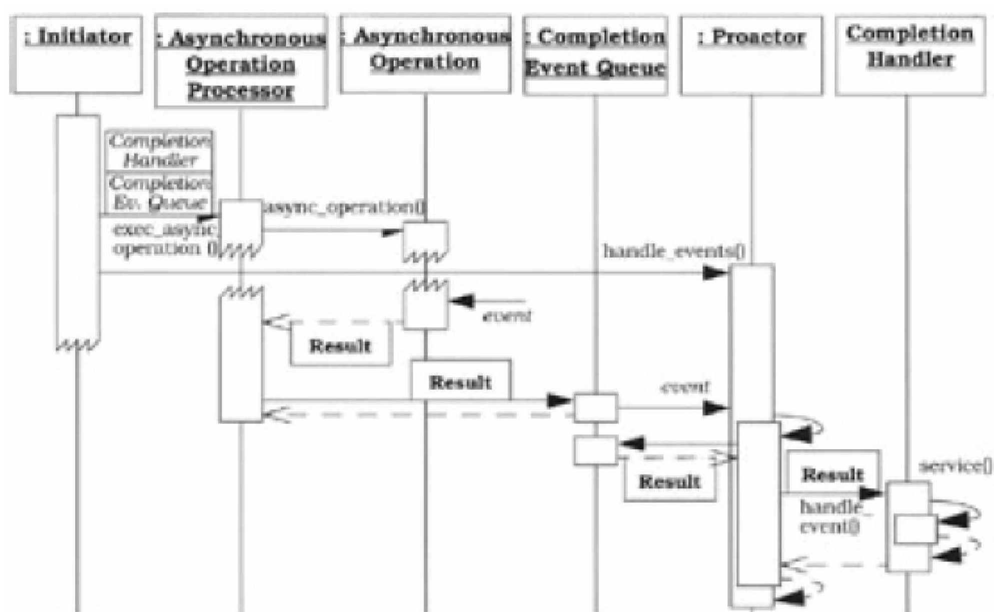
Iniciatorius yra sistemos lokali esybė, kuri iškviečia asinchronines operacijas asinchroninio operacijų procesoriaus pagalba. Iniciatorius dažnai apdoroja kviečiamų asinchroninių operacijų rezultatus, todėl jis taip pat laikomas konkrečiu užbaigimo apdorotoju.

Klasių diagrama yra nurodyta sekančiame paveiksle: pav. 13.



pav. 13 Proaktoriaus schemos klasių diagrama [SSH00+].

Sekos diagrama yra nurodyta sekančiame paveiksle: pav. 14.



pav. 14 Proaktoriaus schemos sekos diagrama [SSH00+].

1.2.2.1. Proaktoriaus schemos variantai

§ Vienos gijos ir grįžtamojo kvietimo modelis.

Vienoje gijoje sinchroniškai yra užmezgami prisijungimai, o užklausų nuskaitymai ir vykdymai yra atliekami asinchroniškai prieš tai užregistravus grįžtamuosius kvietimus asinchroninių operacijų procesoriuje.

§ Atskiros gijos ryšio užmezgėjui, gijos prisijungimo eilės apdorojimui ir asinchroniškų užklausos apdorotojų su grįžtamuoju kvietimu modelis.

Vienoje gijoje yra užmezgami prisijungimai, kurie vėlai yra įterpiami į prisijungimų eilę. Kitoje gijoje sulaukus prisijungimus, užklausų nuskaitymai ir vykdymai yra atliekami asinchroniškai prieš tai užregistravus grįžtamuosius kvietimus asinchroninių operacijų procesoriuje.

§ Vienos gijos modelis su parametrizuota gijų grupe pasibaigusių asinchroninių operacijų rezultatų apdorojimui.

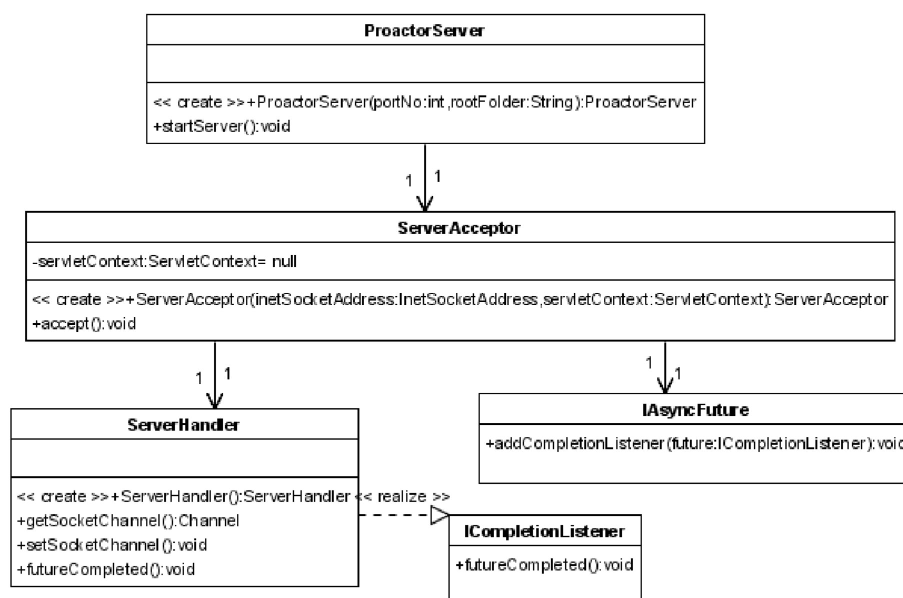
Vienoje gijoje sinchroniškai yra užmezgami prisijungimai, o užklausų nuskaitymai ir vykdymai yra atliekami asinchroniškai. Pasibaigusių asinchroninių operacijų rezultatai yra apdorojami įvykių apdorotojais laukiančiais minėtus įvykius eilėje.

1.2.2.2. Modelio realizacija.

Nors nuo JAVA 1.4 versijos atsirado papildomos galimybės vykdyti nesiblokuojančias įvedimo ir išvedimo operacijas, vis dar egzistuoja tam tikri apribojimai. Visų pirmą vykdyti asinchronines rašymo ir skaitymo operacijas galima tik panaudojant tinklinių susijungimų kanalus. Vykdant tas pačias operacijas skaitant ar rašant bylas, procesas užsiblokuoja tol kol operacija nėra iki galo įvykdoma. Sekantis apribojimas yra tas, kad praktiškai nėra galimybės sužinoti ar asinchroninė operacija baigė savo vykdymą. Todėl įskaitant šiuos nepatogumus, tikros proaktoriaus schemos realizavimas JAVA programavimo kalbos priemonėmis nėra galimas. Galimas yra tik šitos schemos variantas, kur asinchroninių operacijų vykdymą galima realizuoti gijų pagalba.

Kaip ir daugelio JAVA problemų atvejų į pagalba ateina JNI interfeisas. Iš tikrųjų, 2004 metais IBM Alphaworks sukūrė priemones, kurios leidžia vykdyti 100% asinchroninės įvedimo ir išvedimo operacijas kaip su tinkliniais prisijungimo kanalais, taip ir su bylomis. Kadangi operacijos bus vykdomos ne tiesiogiai, t.y. nepanaudojant „gimtojo“ JAVA programavimo kalbos asinchroninių operacijų procesoriaus, D. Schmidto proaktoriaus schema bus pakoreguota atsižvelgiant į IBM AIO4J bibliotekos teikiamą funkcionalumą.

Realizuojamo JAVA kalboje modelio struktūra yra nurodyta paveiksle pav. 15.

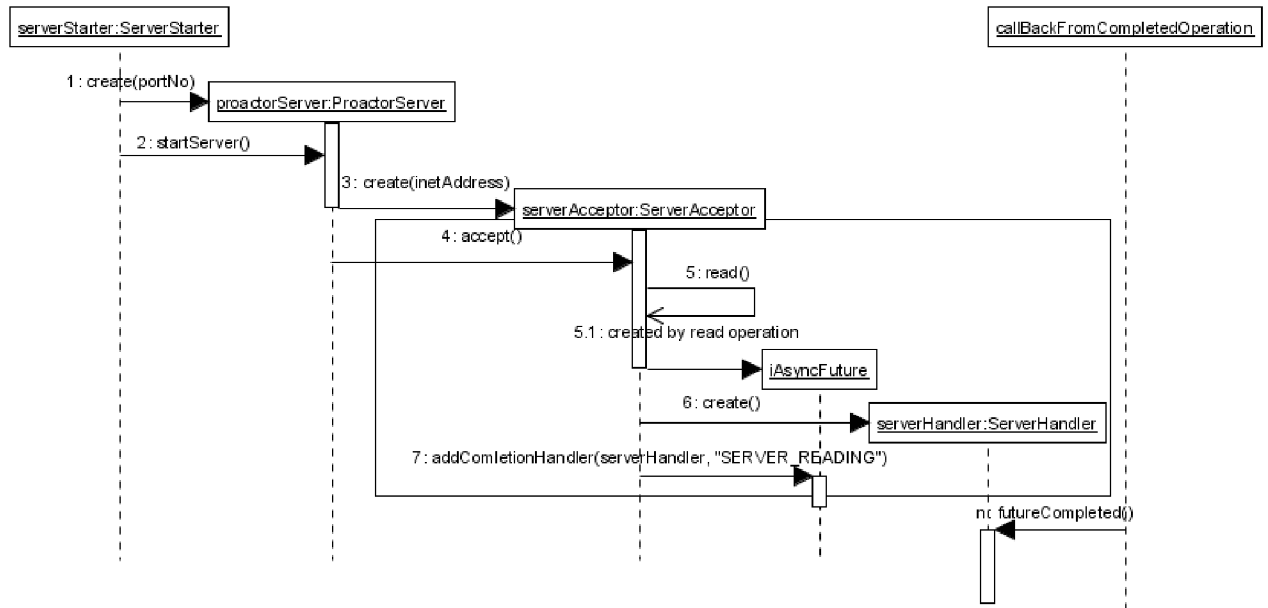


pav. 15 Proaktoriaus schemos pagrindu realizuojamo konkurentiškumą valdančio modelio klasių diagrama.

Klasių aprašymai:

1. *ProactorServer* – klasė atsakinga už proaktoriaus schemos konkurentiškumo mechanizmo paleidimą.
2. *ServerAcceptor* – klasė atsakinga už ryšio užmezgimo ir už užklausų apdorotojų sukūrimą.
3. *ServerHandler* – klasė atsakinga už užklausų apdorojimą.

Modelio veikimas yra parodytas sekos diagramoje pav. 16.



pav. 16 Proaktoriaus schemos pagrindu realizuojamo konkurentiškumą valdančio modelio sekos diagrama.

Pavaizduotojoje aukščiau scenarijaus diagramoje galima pamatyti kaip ir kokia tvarka sąveikauja tarpusavyje minėtų klasių objektai.

- Programa prasideda nuo proaktoriaus serverio paleidimo.
- Serveris sukuria ryšio užmezgėjo objektą.
- Serveris paleidžia pagrindinį proaktoriaus ciklą, kurio kiekvienos iteracijos metu yra prašoma ryšio užmezgėjo užmėgsti prisijungimą.
- Jeigu ryšio užmezgėjo laukimo eilėje yra laukiančiųjų klientų, užmezgamas yra ryšis ir iškarto įvykdoma asinchroninė skaitymo operacija, kuri grąžiną *IAsyncFuture* objektą. *IAsyncFuture* objekto rolę galima sutapatinti su proaktoriaus objektu minėtu D. Schimdt'o schemoje. Ryšio užmezgėjas sukuria užklausą apdorojantį objektą, kuris yra užregistruojamas *IAsyncFuture* objekte ir vėl pereiną į būseną galinančią patvirtinti kitų klientų ryšio užmezgimo norus.
- Užregistravus „proaktoriuje" užklausą apdorojantį objektą, operacijos užbaigimo metu, bus įvykdytas *futureCompleted* metodas esantis užklausos apdorojančiame objekte ir tokiu būdu apdoroti asinchroniškai įvykdytos operacijos rezultatai. Vykdomas rezultatų apdorojimo

veiksmus, objektas vėl gali įvykdyti asinchronines skaitymo ar rašymo operacijas užregistruojant atitinkamus objektus „proaktoriuje“.

- Gijų kiekis realizuotame modelyje visada išlieka pastovūs ir lygus vienam.

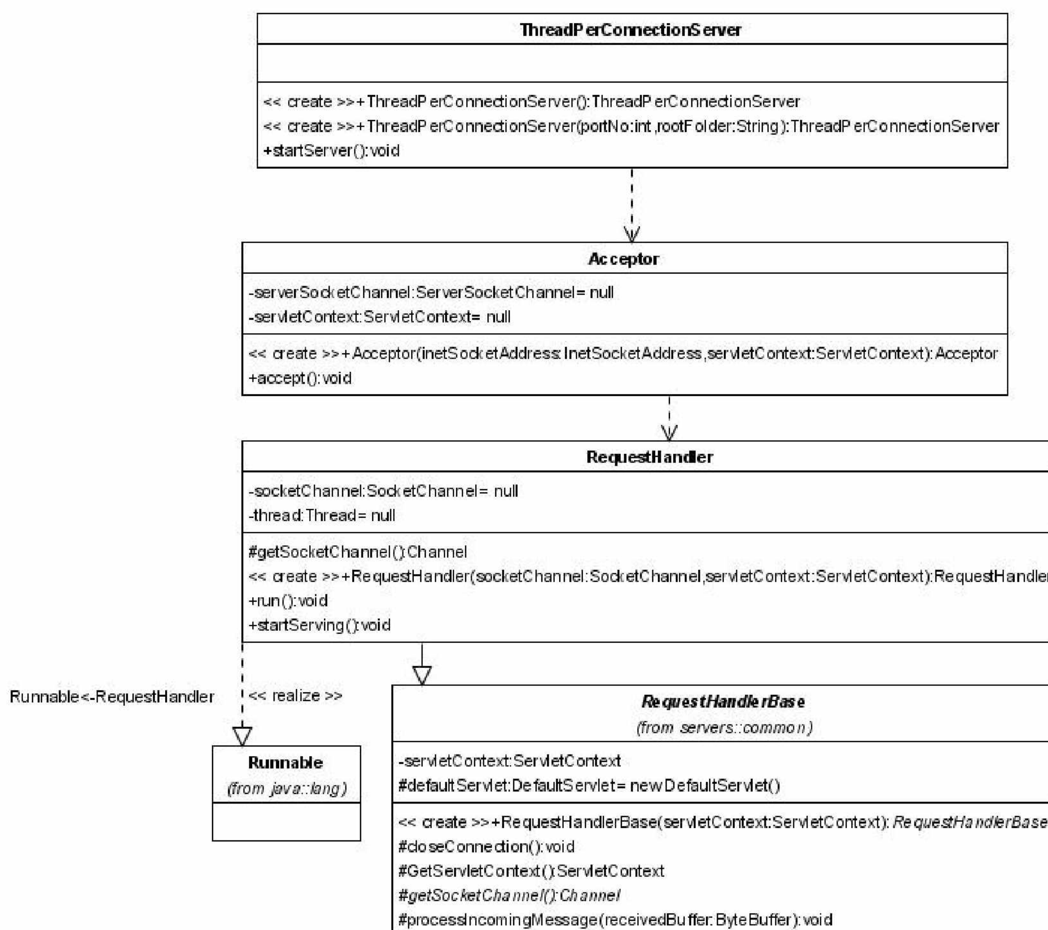
2. Bazinių konkurentiškumo valdymo modelių modifikacijos.

2.1. Sinchroninis besiblokuojantis užklausų apdorojimas.

2.1.1. Vienos gijos vienam prisijungimui modelis.

Šitas modelis yra iš esmės aprašyto skyriuje 1.1 gijos pagrindu valdomo konkurentiškumo modelio atkartojimas, išskirtinių pakeitimų neturi ir sudaro bazę sekančioms modelio modifikacijoms. Nežiūrint į tai kad, pasikeitė bendras visoms realizacijoms užklausų vykdymo komponentas, šio modelio principas išlieka tas pats, t.y. kiekvienos užklausos apdorojimas vyksta sinchroniškai besiblokuojant dedikuotojoje prisijungimui atskiroje gijoje.

Realizuojamo modelio struktūra yra nurodyta paveiksle pav. 17.

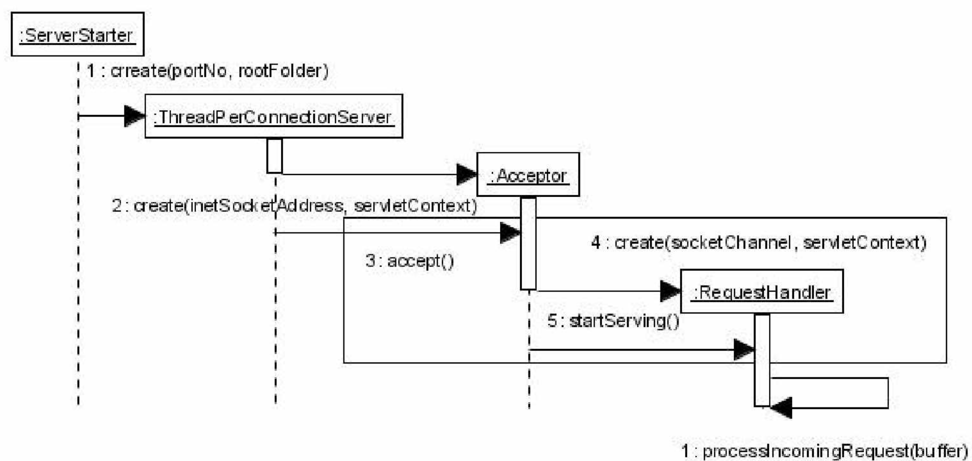


pav. 17 Vienos gijos vienam prisijungimui modelio klasės diagrama

Klasių aprašymai:

1. *ThreadPerConnectionServer* – pagrindinė modelio klasė atsakinga už ryšio užmezgėjo sukūrimą bei už programos ciklą, kuriame ryšio užmezgėjas bando užmegzti ryšį su klientinėmis aplikacijomis.
2. *Acceptor* – ryšio užmezgėjas. Klasė atsakinga už ryšio užmezgimą su klientinėmis aplikacijomis, naujos gijos sukūrimą, kuriai priskiriamas yra naujai sukurtas užklausų apdorotojas.
3. *RequestHandler* – užklausų apdorotojas. Klasė atsakinga už sinchronišką ir besiblokuojančią užklausos apdorojimą.

Modelio veikimas yra parodytas sekos diagramoje pav. 18.



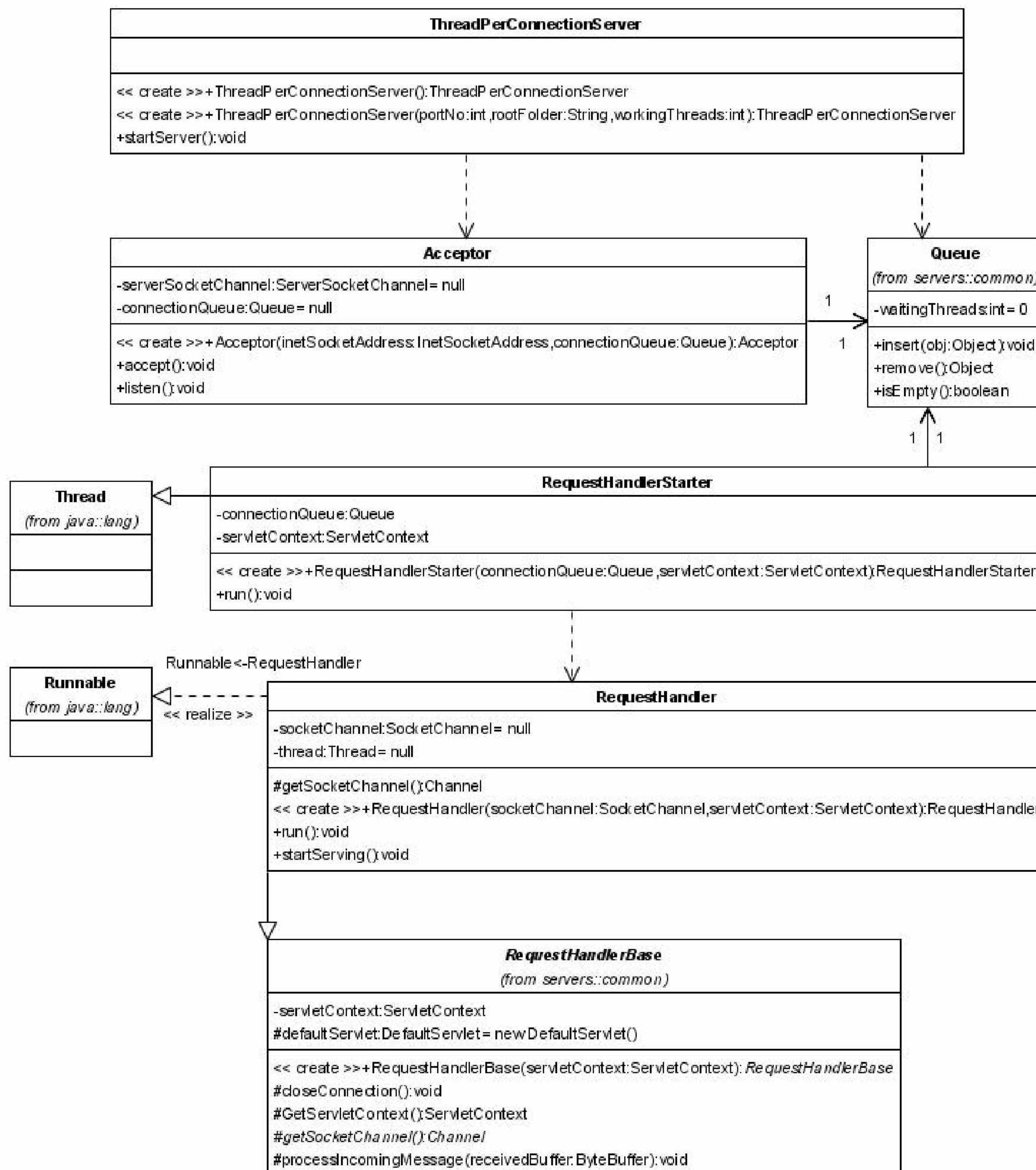
pav. 18 Vienos gijos vienam prisijungimui modelio sekos diagrama.

- Veiksmas prasideda nuo serverio paleidimo. Pirmajame žingsnyje sukuriamas yra *ThreadPerConnectionServer* objektas, kuris valdo pagrindinę programos eigą.
- Antrame žingsnyje sukuriamas yra ryšio užmezgėjo objektas (*Acceptor*) su atitinkamu porto numeriu ir informacijos kontekstu.
- Sukūrus ryšio užmezgėją startuojamas yra programos ciklas. Kiekvienoje ciklo iteracijoje sinchroniškai yra kviečiamas *Acceptor* objekto ryšio užmezgimo metodas, kuris besiblokuodamas laukia naujų ryšio užmėgimo užklausų.
- Sulaukus naujos ryšio užmezgimo užklausos, sukuriamas yra naujas užklausos apdorotojo objektas (*Requesthandler*) .
- Sekančiu žingsniu yra kviečiamas metodas, kuris sukuria naują giją ir sukurtoje gijoje savarankiškai pradeda apdoroti užklausą. Tuo tarpu pagrindinė programos eigos kontrolė grąžinama yra ciklui, kuriame ryšio užmėgėjas vėl laukia prisijungimo užklausų ir viskas pradeda veikti tuo pačiu būdu.
- Kiekviena naujai sukurta gija užbaigus užklausos apdorojimą yra panaikinama.

2.1.2. Vienos gijos vienam prisijungimui modelis su prisijungimo eile.

Šito modelio skirtumą nuo aukščiau minėto sudaro prisijungimo eilės buvimas. Tokiu būdu yra atskiriamas ryšio užmezgėjas nuo užklausos apdorotojo, kas teoriškai turėtų sumažinti atmetamų prijungimo užklausų kiekį. Taip yra manoma todėl, kad naujos gijos sukūrimas sudaro tam tikrą laiko tarpą, per kurį susikaupus ribiniam prisijungimo užklausų kiekiui operacinės sistemos lygmenyje būtų numetinėjamos visos naujai ateinančios prisijungimo užklausos.

Realizuojamo modelio struktūra yra nurodyta paveiksle pav. 19.

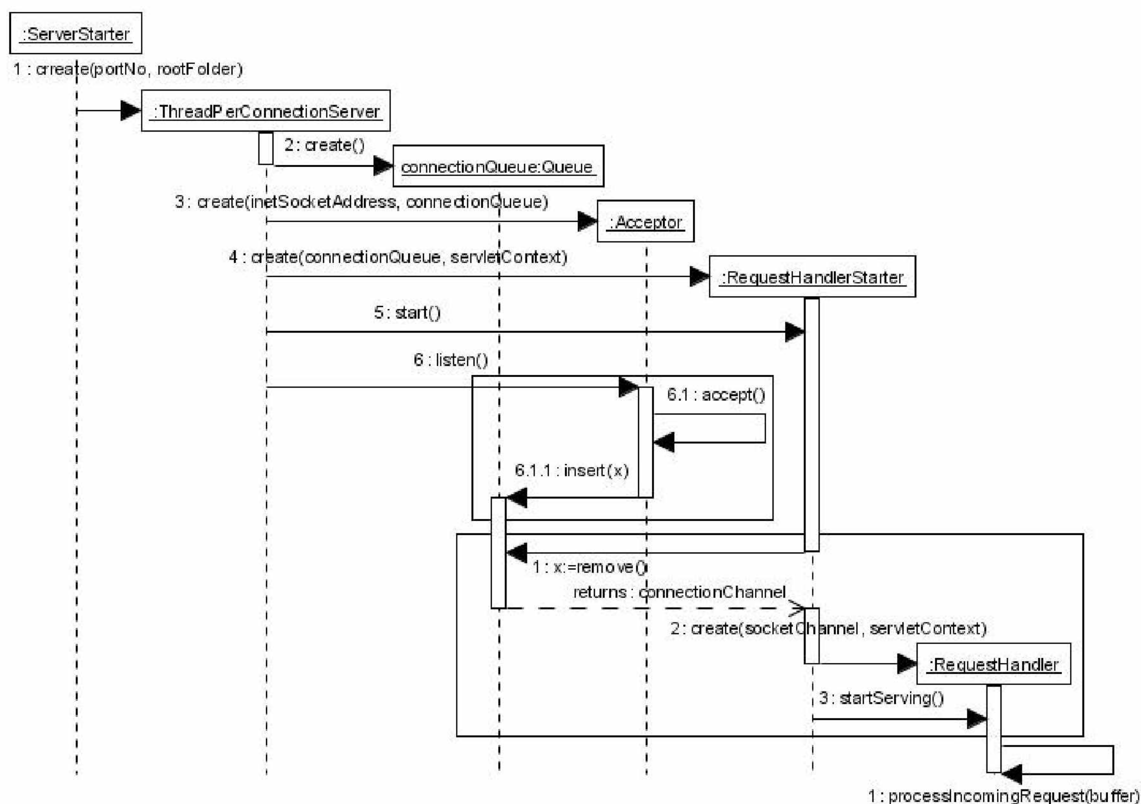


pav. 19 Vienos gijos vienam prisijungimui modelis su prisijungimo eile klasių diagrama

Klasių aprašymai:

1. *ThreadPerConnectionServer* – pagrindinė modelio klasė atsakinga už ryšio užmezgėjo bei tuščios prisijungimo eilės sukūrimą, užklausų apdorotojų sukūrėjo startavimą ir už ryšio užmezgėjo ciklo paleidimą, kuriame bandoma užmegzti ryšį su klientinėmis aplikacijomis.
2. *Acceptor* – ryšio užmezgėjas. Klasė atsakinga už ryšio užmezgimą su klientinėmis aplikacijomis bei prisijungimų pridėjimą prie bendros prisijungimo eilės.
3. *RequestHandlerStarter* – užklausų apdorotojų sukūrėjas. Klasė atsakinga už užklausų apdorotojų sukūrimą naudojant bendrą prisijungimo eilę.
4. *RequestHandler* – užklausų apdorotojas. Klasė atsakinga už sinchronišką ir besiblokuojančią užklausos apdorojimą.
5. *Queue* – prisijungimų eilė. Klasė atsakinga už prisijungimų kaupimą.

Modelio veikimas yra parodytas sekos diagramoje pav. 23.



pav. 20 Vienos gijos vienam prisijungimui modelis su prisijungimo eile sekos diagrama

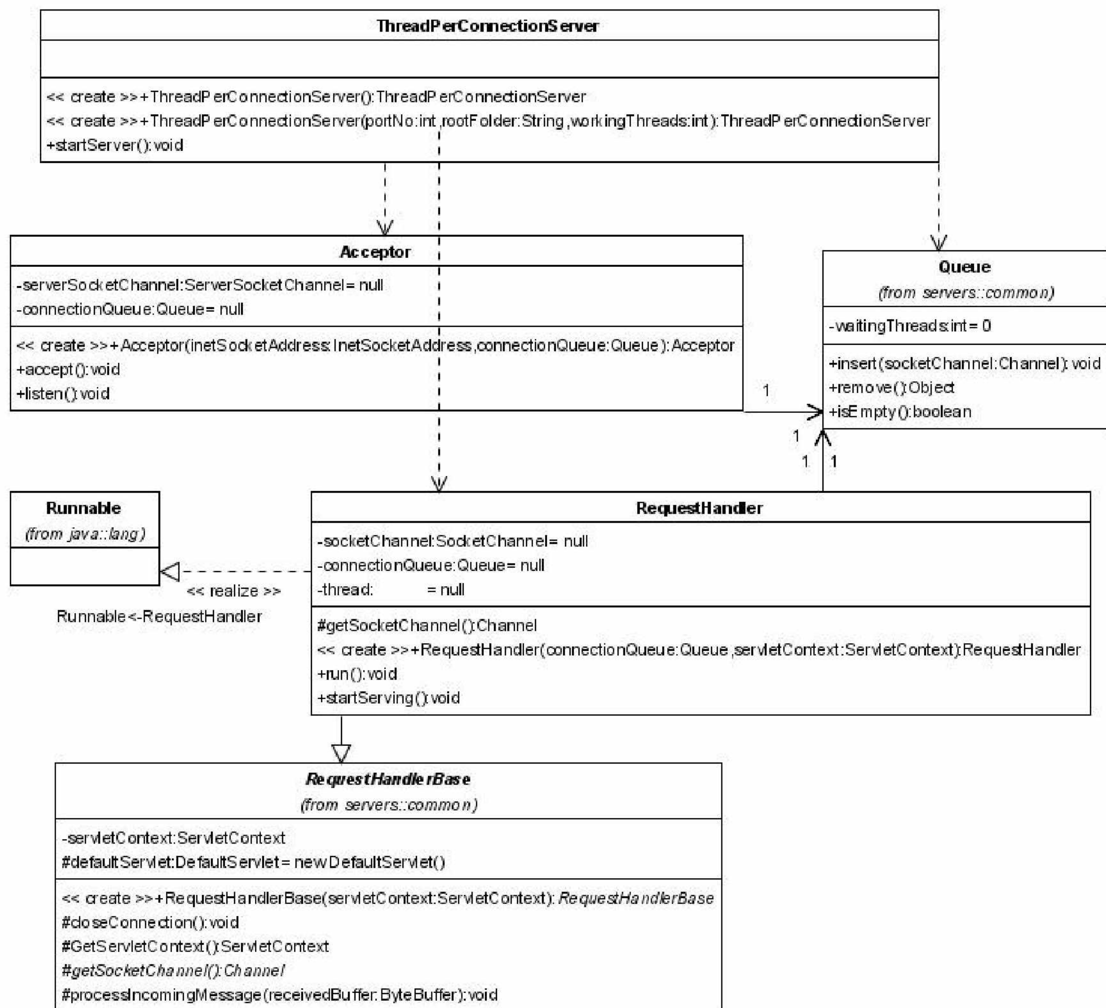
- Veiksmas prasideda nuo serverio paleidimo. Pirmajame žingsnyje sukuriamas yra *ThreadPerConnectionServer* objektas, kuris valdo pagrindinė programos eigą.
- Antrame žingsnyje yra sukuriama tuščia bendra visiems suinteresuotiems objektams prisijungimų eilė (*connectionQueue*).
- Trečiame žingsnyje yra sukuriamas ryšio užmezgėjo objektas (*Acceptor*) su atitinkamu porto numeriu.
- Ketvirtame žingsnyje yra sukuriamas užklausų apdorotojų sukūrėjo objektas (*RequestHandlerStarter*) su atitinkamu informacijos kontekstu.

- Penktame žingsnyje yra sukuriama atskira gija užklausų apdorotojų sukūrėjui. Pagrindinė programos eiga vykdo šeštą žingsnį, tuo tarpu savarankiškai veikiančioje gijoje minėtas objektas amžino ciklo kiekvienos iteracijos metu užsiblokuoja laukdamas prisijungimų bendroje prisijungimų eilėje.
  - Atsiradus naujam prisijungimui yra sukuriamas užklausų apdorotojas (*Requesthandler*).
  - Sekančiu žingsniu yra kviečiamas užklausų apdorotojo metodas *startServing*, kuris sukuria naują giją ir sukurtoje gijoje savarankiškai pradeda apdoroti užklausą.
  - Kiekviena užklausų apdorotojui naujai sukurta gija užbaigus užklausos apdorojimą yra panaikinama.
- Šeštame žingsnyje yra kviečiamas ryšio užmezgėjo objekto metodas *listen*, kuris sukdamas viduje amžiną ciklą užsiblokuoja laukdamas naujų prisijungimo užklausų. Atsiradus naujai prisijungimo užklausai yra užmezgamas ryšis ir prisijungimas pridedamas prie prisijungimo eilės.

2.1.3. Vienos gijos vienam prisijungimui modelis su prisijungimo eile ir parametrizuota sukurtu gijų grupe.

Šis modelis yra aprašyto praeitame skyriuje vienos gijos vienam prisijungimui modelio su prisijungimo eile modifikacija. Priešingai negu aukščiau aprašyto modelio atveju, kur atsiradus prisijungimui bendroje prisijungimų eilėje kiekvienas užklausų apdorotojas pradėdavo veikti naujai sukurtoje gijoje, šiame modelyje gijos nėra dinamiškai sukuriamos. Konkurentiškumą valdantis komponentas yra parametrizuojamas darbinių gijų skaičiumi, kurios yra sukuriamos startuojant serverio darbui ir bendrai naudojamos užklausų apdorotojų. Tokiu būdu teoriškai yra tikimasi sumažinti bendra užklausos apdorojimą laiką. Taip yra manoma todėl, kad gijos sukūrimas sudaro tam tikrą laiko tarpą, kuris prie tam tikros apkrovos gali būti ilgesnio užklausų apdorojimo priežastimi. Modelio galimą nepatogumą gali sudaryti darbinių užklausų parametrizavimas, t.y. sunku nuspėti koks darbinių gijų skaičius bus optimalus esant konkrečiam konkurentiškumo valdymo komponento apkrovimui.

Realizuojamo modelio struktūra yra nurodyta paveiksle pav. 24.

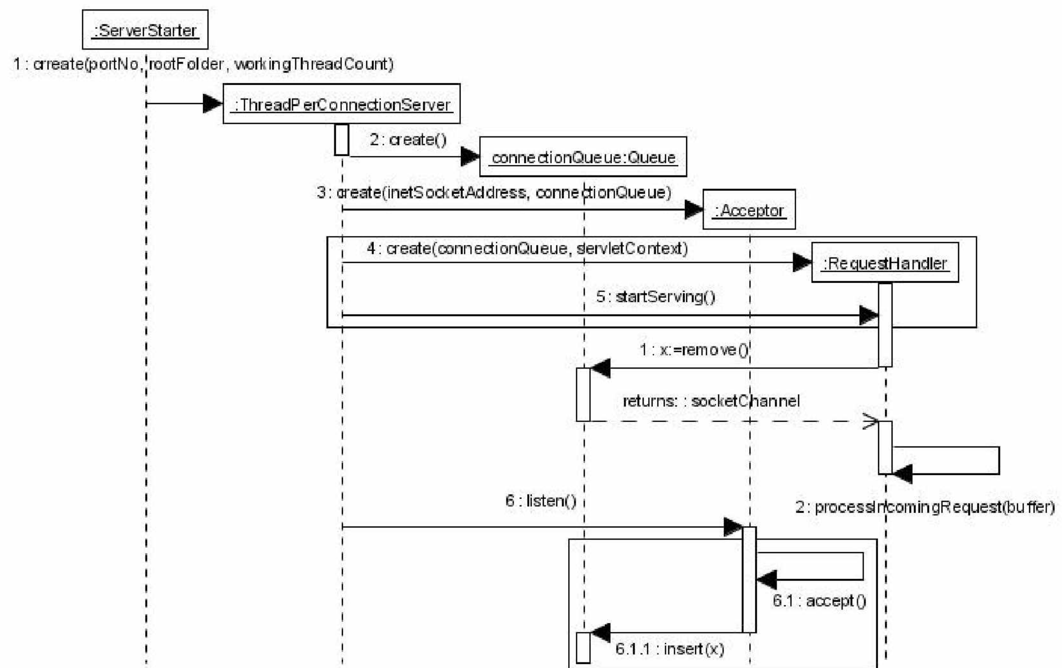


pav. 21 Vienos gijos vienam prisijungimui modelio su prisijungimo eile ir parametrizuota sukurtu gijų grupe klasių diagrama

Klasių aprašymai:

1. *ThreadPerConnectionServer* – pagrindinė modelio klasė atsakinga už ryšio užmezgėjo bei tuščios prisijungimo eilės sukūrimą, darbinių gijų su užklausų apdorotojais sukūrimą bei paleidimą ir už ryšio užmezgėjo ciklo paleidimą, kuriame bus bandoma užmegzti ryšį su klientinėmis aplikacijomis.
2. *Acceptor* – ryšio užmezgėjas. Klasė atsakinga už ryšio užmezgimą su klientinėmis aplikacijomis bei prisijungimų pridėjimą prie bendros prisijungimo eilės.
3. *RequestHandler* – užklausų apdorotojas. Klasė atsakinga už sinchronišką ir besiblokuojančią užklausos apdorojimą.
4. *Queue* – prisijungimų eilė. Klasė atsakinga už prisijungimų kaupimą.

Modelio veikimas yra parodytas sekos diagramoje pav. 25.



pav. 22 Vienos gijos vienam prisijungimui modelio su prisijungimo eile ir parametrizuota sukurtu gijų grupe sekos diagrama.

- Veiksmas prasideda nuo serverio paleidimo. Pirmajame žingsnyje sukuriamas yra *ThreadPerConnectionServer* objektas, kuris valdo pagrindinė programos eigą.
- Antrame žingsnyje yra sukuriama tuščia bendra visiems suinteresuotiems objektams prisijungimų eilė (*connectionQueue*).
- Trečiame žingsnyje yra sukuriamas ryšio užmezgėjo objektas (*Acceptor*) su atitinkamu porto numeriu.
- Nuo ketvirto žingsnio prasideda ciklas, kuris pagal nurodytą darbinių gijų skaičių sukuria atitinkamo kiekio gijų grupę.
- Sukūrus kiekvieną užklausų apdorotojo (*RequestHandler*) objektą yra kviečiamas to objekto *startServing* metodas, kuris pradeda darbą vienoje iš sukurtų gijų užsiblokuodamas laukdamas prisijungimų bendroje prisijungimų eilėje, tuo tarpu pagrindinė programos vykdymo eiga grįžta prie šešto žingsnio.
    - Atsiradus naujam prisijungimui yra pradedamas užklausos apdorojimas. Einamoji užklausos apdorotojo gija yra pažymimą užimtą ir nereaguojančią į naujus prisijungimus.
    - Užbaigus užklausos apdorojimą giją yra pažymima laisva ir reaguojančią į naujus prisijungimus.
- Šeštame žingsnyje yra kviečiamas ryšio užmezgėjo objekto metodas *listen*, kuris sukdamas viduje amžiną ciklą užsiblokuoja laukdamas naujų prisijungimo užklausų. Atsiradus naujai

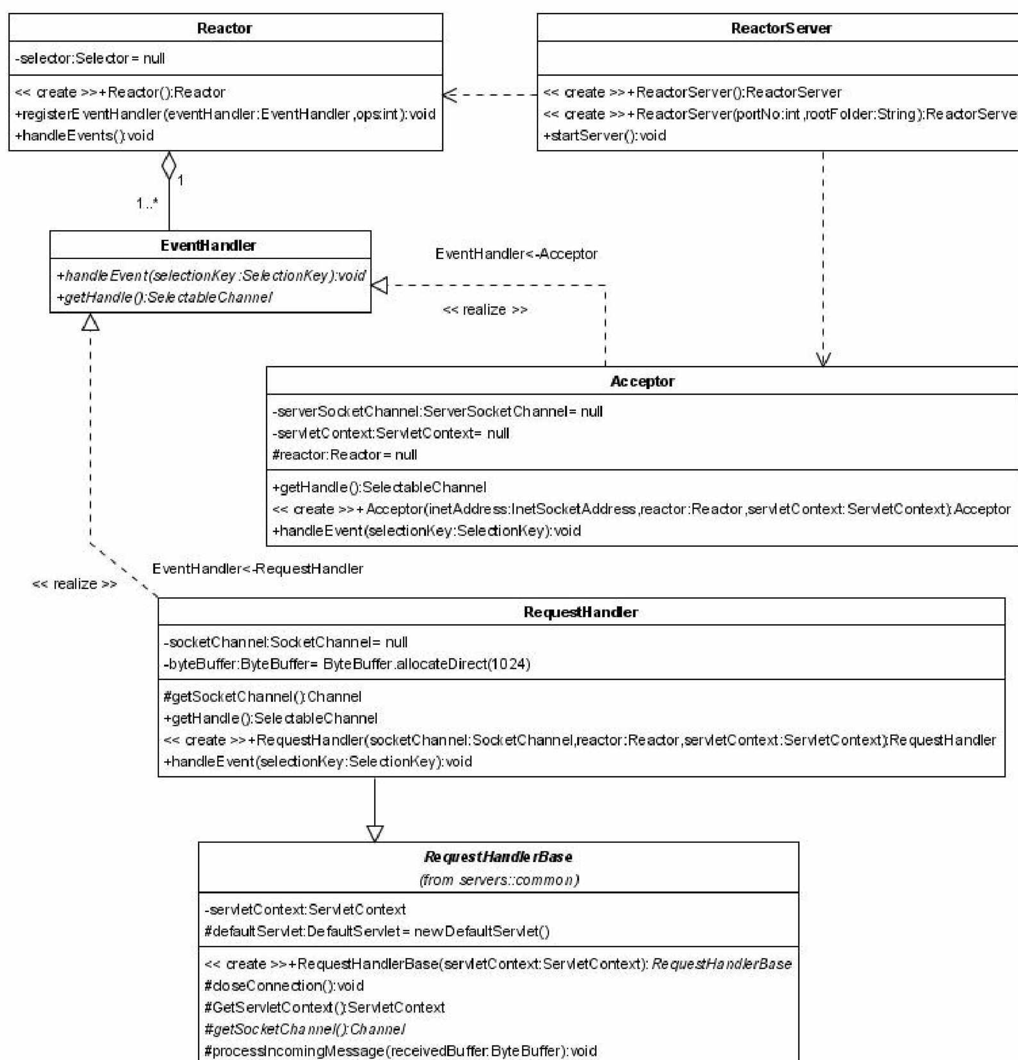
prisijungimo užklausai yra užmezgamas ryšis ir prisijungimas pridedamas prie bendros prisijungimo eilės.

2.2. Sinchroninis nesiblokuojantis užklausų apdorojimas. Reaktoriaus schemos variantai.

2.2.1. Vienos gijos ir vieno įvykių išskyrėjo modelis.

Šitas modelis yra iš esmės aprašyto skyriuje 1.2.1 reaktoriaus schemos pagrindu valdomo konkurentiškumo modelio atkartojimas ir išskirtinių pakeitimų neturi. Aprašyta žemiau modelio struktūra ir veikimas sudaro bazę sekančioms modelio modifikacijoms. Modelio principas išliko tas pats, t.y. ryšių užmezgėjas ir užklausų apdorotojai yra užregistruojami vienoje įvykių išskyrėjo eilėje. Įvykių išskyrimas ir apdorojimas bei pačių užklausų apdorojimas yra vykdomi sinchroniškai besiblokuodami toje pačioje gijoje.

Realizuojamo modelio struktūra yra nurodyta paveiksle pav. 23.

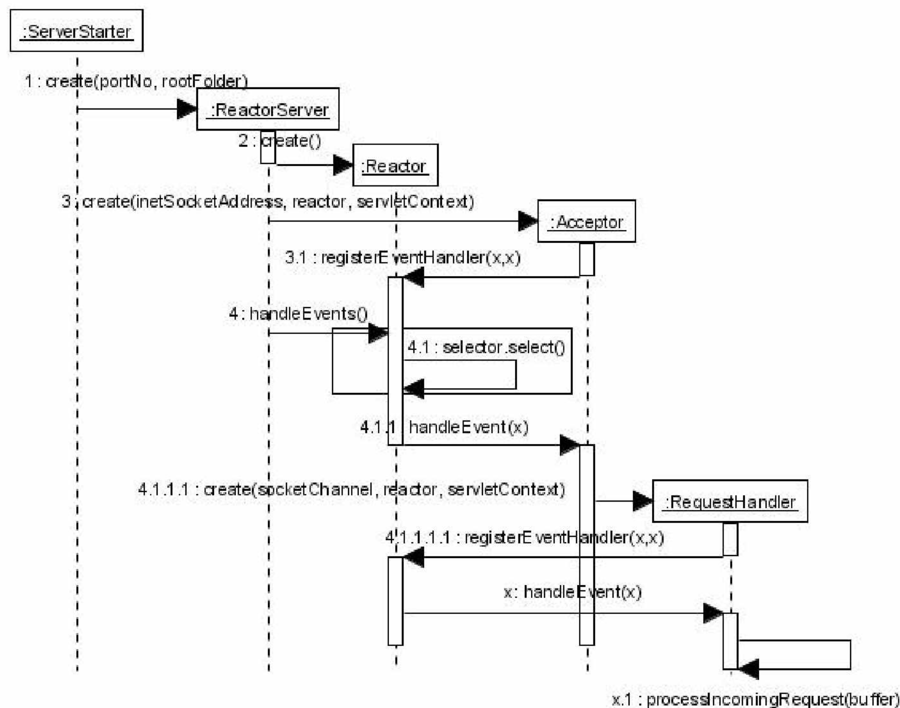


pav. 23 Vienos gijos ir vieno įvykių išskyrėjo modelio klasės diagrama

Klasių aprašymai:

1. *ReactorServer* – pagrindinė modelio klasė atsakinga už reaktoriaus bei ryšio užmezgėjo objektų sukūrimą bei reaktoriaus darbo paleidimą.
2. *Reactor* – reaktorius. Klasė atsakinga už įvykių išskyrėjo darbą bei už informavimą atitinkamų įvykių apdorotojų.
5. *EventHandler* – įvykio apdorotojas. Bazinė klasė, iš kurios yra paveldėtos ryšio užmezgimo bei skaitymo/rašymo įvykių apdorotojai.
6. *Acceptor* – ryšio užmezgėjas. Klasė reaguojanti į ryšio užmezgimo įvykius. Atsakinga už ryšio užmezgimą bei už užklausų apdorotojų sukūrimą.
7. *RequestHandler* – užklausų apdorotojas. Klasė reaguojanti į skaitymo/rašymo įvykius. Atsakinga už užklausų apdorojimą.

Modelio veikimas yra parodytas sekos diagramoje pav. 27.



pav. 24 Vienos gijos ir vieno įvykių išskyrėjo modelio scenarijaus diagrama

- Veiksmas prasideda nuo serverio paleidimo. Pirmajame žingsnyje sukuriamas yra *ReactorServer* objektas, kuris valdo pagrindinę programos eigą.
- Antrame žingsnyje yra sukuriamas reaktoriaus objektas (*Reactor*).
- Trečiame žingsnyje yra sukuriamas ryšio užmezgėjo objektas (*Acceptor*).). Kuriant objektą parametrais yra perduodami prisijungimo adreso, reaktoriaus bei informacijos konteksto objektai.
    - Sukūrus ryšio užmezgėjo objektą, registruojame jo suinteresuotumą prisijungimo įvykiais reaktoriaus objekte esančiame įvykio išskyrėjuje.

- Ketvirtame žingsnyje kviečiamas yra reaktoriaus metodas *handleEvents*, kuris startuoja amžiną ciklą. Kiekvienos iteracijos metu sinchroniškai yra atliekami tokie veiksmai:
  - Įvykio išskyrėjas pradedama tikrinti ar neatsirado prisijungimo/skaitymo/rašymo įvykių. Aptikus įvykį tikrinimas yra nutraukiamas ir pereinama prie kito žingsnio.
  - Atsiradus prisijungimo įvykiui ryšio užmezgėjas atitinkamai sureaguoja ir sukuria užklausos apdorotojo objektą (*RequestHandler*). Kuriant objektą parametrais yra perduodami prisijungimo, reaktoriaus bei informacijos konteksto objektai.
    - Sukūrus užklausos apdorotojo objektą, registruojame jo suinteresuotumą skaitymo/rašymo įvykiais reaktoriaus objekte (*reactorForRequestHandler*) esančiame įvykio išskyrėjuje.
  - Atsiradus skaitymo/rašymo įvykiui užklausų apdorotojas atitinkamai sureaguoja ir sinchroniškai nesiblokuojant apdoroja užklausą.

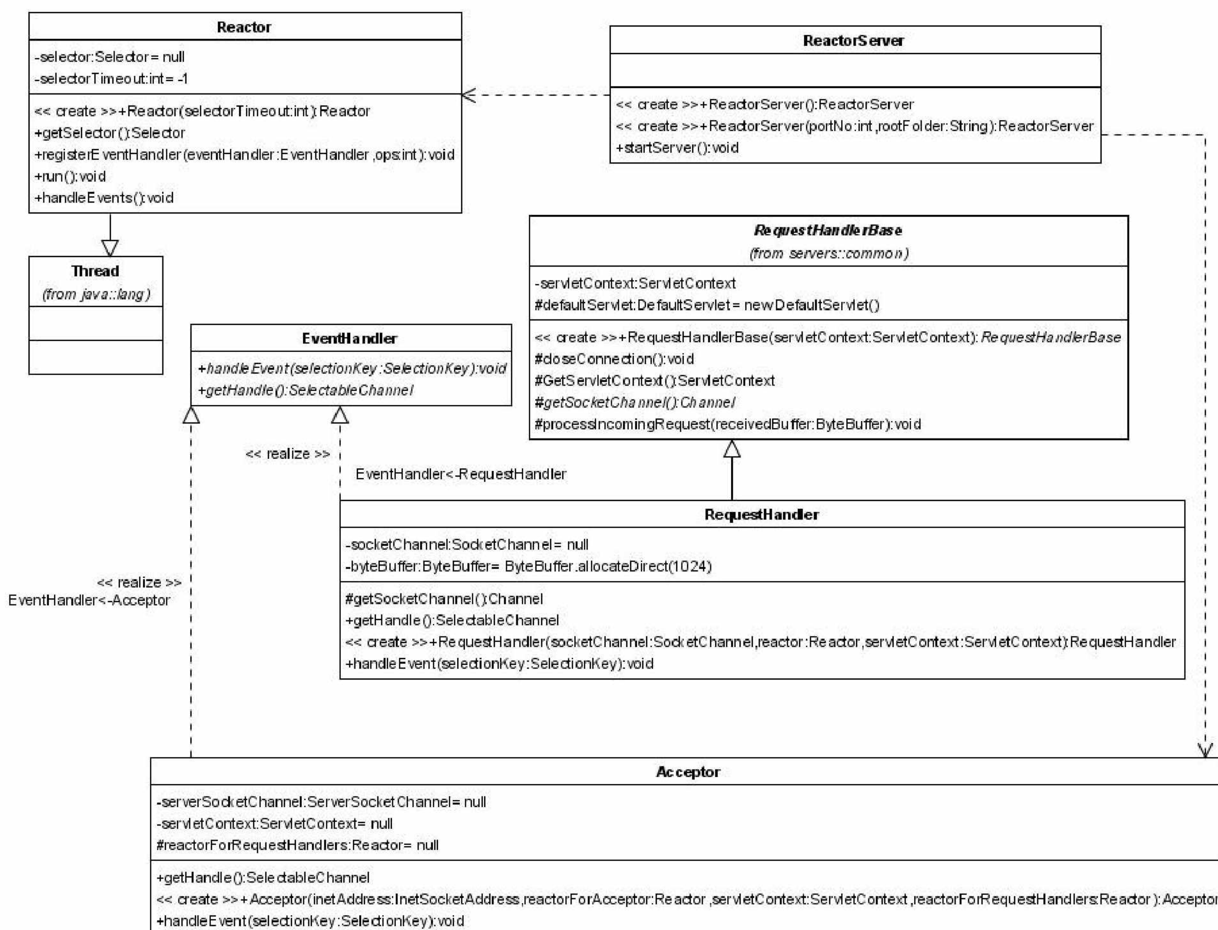
2.2.2. Atskiros gijos su įvykių išskyrėju ryšio užmezgėjui ir gijos užklausų apdorotojams modelis.

Šito modelio skirtumą nuo praeitame skyriuje aprašyto sudaro dviejų savarankiškų gijų buvimas su atskirais įvykių išskyrėjais. Tokios modifikacijos nagrinėjimas buvo sąlygotas tuo, kad bazinio modelio atveju ryšio užmezgimas ir užklausų apdorojimai buvo vykdomi sinchroniškai, vienas po kito. Teoriškai atskirus ryšio užmezgimą ir užklausų apdorojimo veiksmus panaudojant dvi atskiras gijas turėtų sumažėti prisijungimų atmetimų skaičius. Taip yra manoma, nes galimas yra atvejis, kad tam tikros užklausos apdorojimas gali užtrukti tiek laiko, per kurį susikaupus ribiniam prisijungimo užklausų kiekiui operacinės sistemos lygmenyje bus numetinėjamos visos naujai ateinančios prisijungimo užklausos. Iš kitos pusės realizuojant šį konkurentiškumą valdantį modelį atsiranda klausimas kaip ilgai nepertraukiant reikia tikrinti ar įvykių išskyrėjas neaptiko naujų įvykių. Šitoje vietoje galima išskirti keletą variantų:

a) Abu įvykių išskyrėjai užsiblokuoja laukdami įvykių. Kadangi abu įvykių išskyrėjai užsiblokuoja laukdami įvykių, o be to pagal JAVA [SUN02] specifikaciją šitie objektai yra gijų atžvilgių saugūs, šitas variantas 100% neveiks. Taip yra todėl, kad norėdami pridėti prie įvykio iškyrėjo naują užklausų apdorotoją, einamoji gija užsiblokuos laukdama kol įvykių išskyrėjas baigs įvykių atsiradimo laukimą, kas savo ruožtu atsitiks tik jeigu įvykių išskyrėjas prieš tai turėjo užregistruota įvykių apdorotoją, t.y. programos vykdymas atsidurs aklavietėje (*dead-lock*). Dėl minėtos priežasties šitas variantas toliau nebus nagrinėjamas.
b) Abu įvykių išskyrėjai užsiblokuoja laukdami įvykių su vienodu laukimo nutraukimo laiku.

c) Kadangi įvykių išskyrėjas dedikuotas ryšio užmezgėjui turi tik viena prisijungimų įvykių generatorių prasminga nustatyti, kad jis visada galėtų užsiblokuoti laukdamas naujų prisijungimų įvykių. Tuo tarpu dedikuotas užklausų apdorotojams įvykių išskyrėjas turi būti parametrizuojamas kaip ir b) punkto atveju, t.y. reikia nurodyti laiko tarpą, per kurį bus nepertraukiamai laukiama įvykių.

Realizuojamo modelio struktūra yra nurodyta paveiksle pav. 25.



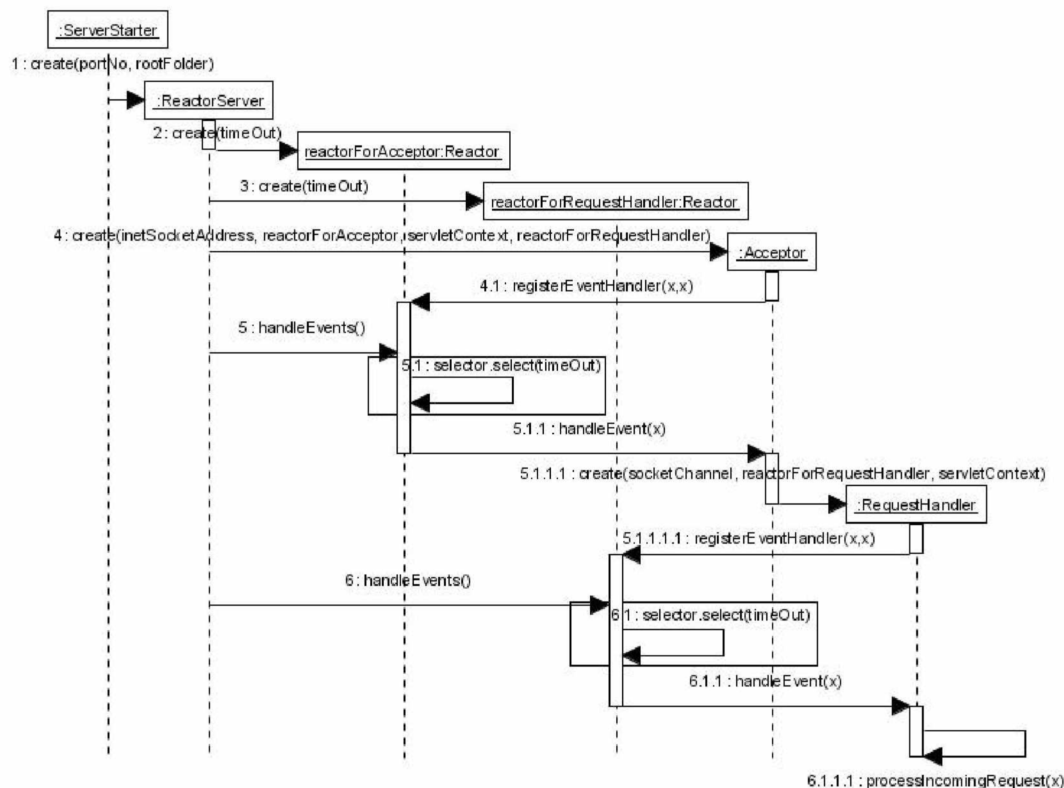
pav. 25 Atskiros gijos su įvykių išskyrėju ryšio užmezgėjui ir gijos užklausų apdorotojams modelio klasių diagrama

Klasių aprašymai:

1. *ReactorServer* – pagrindinė modelio klasė atsakinga už dviejų reaktorių bei ryšio užmezgėjo objektų sukūrimą bei abiejų reaktorių darbo paleidimą.
2. *Reactor* – reaktorius. Klasė atsakinga už įvykių išskyrėjo darbą bei už informavimą atitinkamų įvykių apdorotojų.
3. *EventHandler* – įvykio apdorotojas. Bazinė klasė, iš kurios yra paveldėtos ryšio užmezgimo bei skaitymo/rašymo įvykių apdorotojai.
4. *Acceptor* – ryšio užmezgėjas. Klasė reaguojanti į ryšio užmezgimo įvykius. Atsakinga už ryšio užmezgimą bei už užklausų apdorotojų sukūrimą.

5. *RequestHandler* – užklausų apdorotojas. Klasė reaguojanti į skaitymo/rašymo įvykius. Atsakinga už užklausų apdorojimą.

Modelio veikimas yra parodytas sekos diagramoje pav. 26.



pav. 26 Atskiros gijos su įvykių išskyrėju ryšio užmezgėjui ir gijos užklausų apdorotojams modelio sekos diagrama.

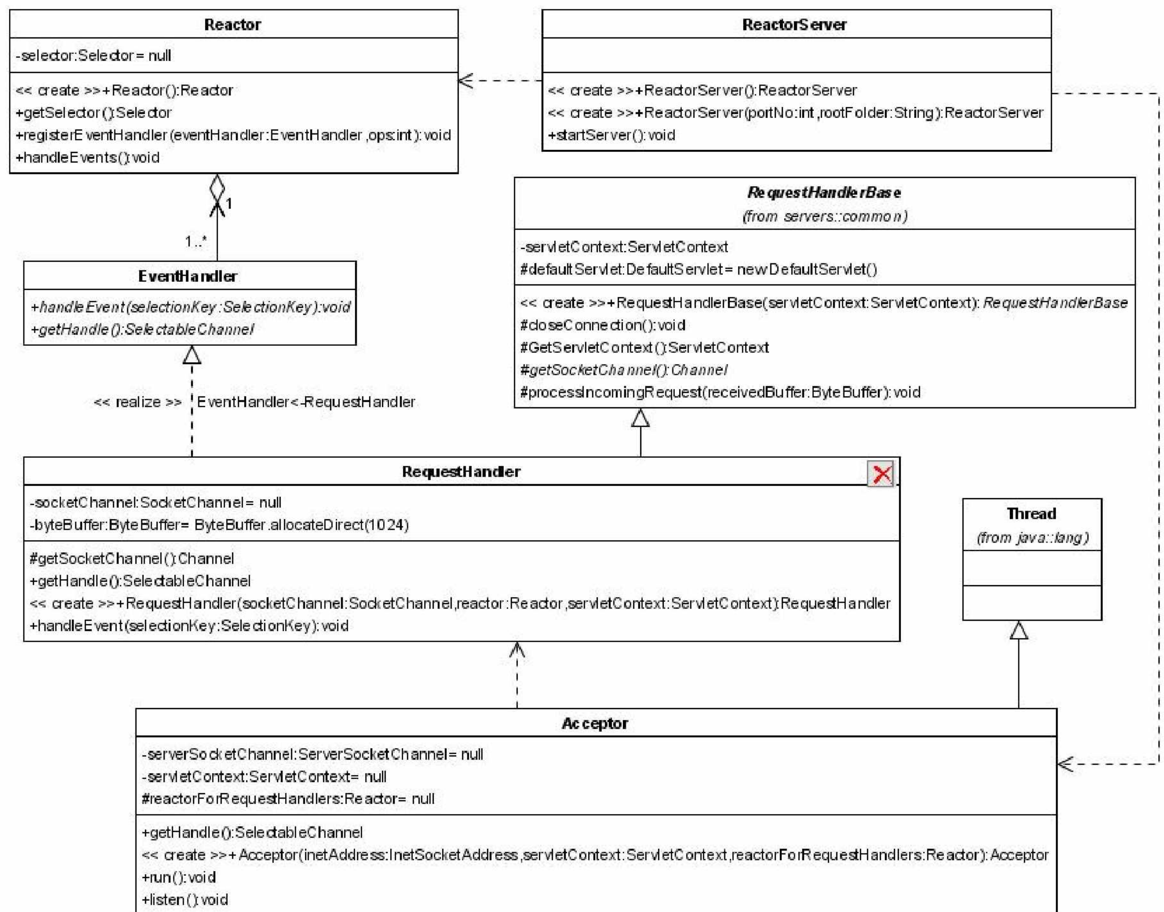
- Veiksmas prasideda nuo serverio paleidimo. Pirmajame žingsnyje sukuriamas yra *ReactorServer* objektas, kuris valdo pagrindinė programos eigą.
- Antrame žingsnyje yra sukuriamas reaktoriaus objektas (*reactorForAcceptor*) skirtas ryšio užmezgėjui. Kuriant objektą parametru yra nurodomas nepertraukiamas laukimo laikas milisekundėmis.
- Trečiame žingsnyje yra sukuriamas reaktoriaus objektas (*reactorForRequestHandler*) skirtas užklausų apdorotojams. Kuriant objektą parametru yra nurodomas nepertraukiamas laukimo laikas milisekundėmis.
- Ketvirtame žingsnyje yra sukuriamas ryšio užmezgėjo objektas (*Acceptor*). Kuriant objektą parametrais yra perduodami prisijungimo adreso, reaktoriaus skirto ryšio užmezgėjui, informacijos konteksto bei reaktoriaus skirto užklausų apdorotojams objektai.
    - Sukūrus ryšio užmezgėjo objektą užregistruojame jo suinteresuotumą prisijungimo įvykiais reaktoriaus objekte objekte (*reactorForAcceptor)* esančiame įvykio išskyrėjuje.

- Penktame žingsnyje kviečiamas yra ryšio užmezgėjui skirto reaktoriaus metodas *handleEvents*, kuris sukūrus savarankišką gija startuoja amžiną ciklą. Kiekvienos iteracijos metu sinchroniškai yra atliekami tokie veiksmai:
  - Pagal nustatytą nepertraukiamo laukimo laiką įvykio išskyrėjas pradedama tikrinti ar neatsirado prisijungimo įvykių. Aptikus įvykį arba pasibaigus laukimo laikui tikrinimas yra nutraukiamas ir pereinama prie kito žingsnio.
  - Atsiradus prisijungimo įvykiui ryšio užmezgėjas atitinkamai sureaguoja ir sukuria užklausos apdorotojo objektą (*RequestHandler*). Kuriant objektą parametrais yra perduodami prisijungimo, reaktoriaus bei informacijos konteksto objektai.
  - Sukūrus užklausos apdorotojo objektą, registruojame jo suinteresuotumą skaitymo/rašymo įvykiais reaktoriaus objekte (*reactorForRequestHandler*) esančiame įvykio išskyrėjuje.
- Šeštame žingsnyje kviečiamas yra užklausų apdorotojams skirto reaktoriaus metodas *handleEvents*, kuris sukūrus savarankišką gija startuoja amžiną ciklą. Kiekvienos iteracijos metu sinchroniškai yra atliekami tokie veiksmai:
  - Pagal nustatytą nepertraukiamo laukimo laiką įvykio išskyrėjas pradedama tikrinti ar neatsirado skaitymo/rašymo įvykių. Aptikus įvykį arba pasibaigus laukimo laikui tikrinimas yra nutraukiamas ir pereinama prie kito žingsnio.
  - Atsiradus skaitymo/rašymo įvykiui užklausų apdorotojas atitinkamai sureaguoja ir sinchroniškai nesiblokuojant apdoroja užklausą.

2.2.3. Atskiros gijos ryšio užmezgėjui ir gijos su įvykio išskyrėju užklausų apdorotojams modelis.

Šito modelio skirtumą nuo praeitame skyriuje aprašyto sudaro tai, kad ryšio užmezgėjo gijoje nėra naudojamas įvykio išskyrėjas. Toks reaktoriaus schemos variantas yra nagrinėjamas dėl to, kad nėra tikslinga turėti užregistruotą viename ar skirtingose įvykių išskyrėjuose ryšio užmezgėją. Taip yra manoma todėl, kad reaktoriaus darbas yra vykdomas sinchroniškai ir įvykių apdorojimas gali užtrukti tiek laiko, per kurį susikaupus ribiniam prisijungimo užklausų kiekiui operacinės sistemos lygmenyje bus numetinėjamos visos naujai ateinančios prisijungimo užklausos. Ryšio užmezgėjas yra realizuojamas kaip atskiroje gijoje veikiantis sinchroninis objektas, kuris užsiblokuoja laukdamas naujų ryšių užmezgimo užklausų. Tuo tarpu dedikuotas užklausų apdorotojams įvykių išskyrėjas turi būti parametrizuojamas nepertraukiamu laukimo laiku.

Realizuojamo modelio struktūra yra nurodyta paveiksle pav. 27.

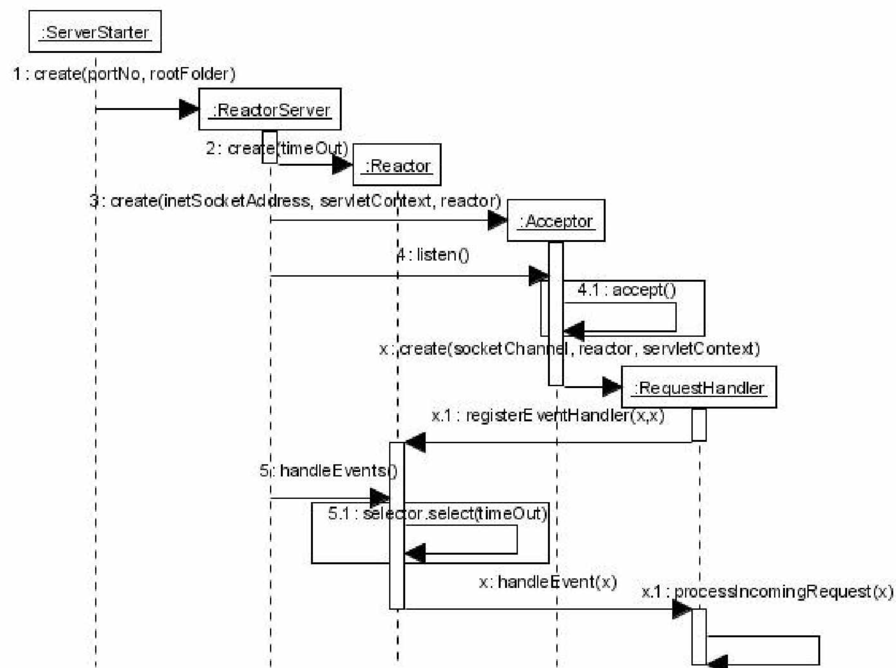


pav. 27 Atskiros gijos ryšio užmezgėjui ir gijos su įvykio išskyrėju užklausų apdorotojams modelio klasių diagrama.

Klasių aprašymai:

1. *ReactorServer* – pagrindinė modelio klasė atsakinga už reaktoriaus bei ryšio užmezgėjo objektų sukūrimą bei abiejų objektų darbo paleidimą.
2. *Reactor* – reaktorius. Klasė atsakinga už įvykių išskyrėjo darbą bei už informavimą atitinkamų įvykių apdorotojų.
3. *EventHandler* – įvykio apdorotojas. Bazinė klasė, iš kurios yra paveldėtas skaitymo/rašymo įvykių apdorotojai.
4. *Acceptor* – ryšio užmezgėjas. Klasė atsakinga už ryšio užmezgimą bei už užklausų apdorotojų sukūrimą.
5. *RequestHandler* – užklausų apdorotojas. Klasė reaguojanti į skaitymo/rašymo įvykius. Atsakinga už užklausų apdorojimą.

Modelio veikimas yra parodytas sekos diagramoje pav. 31.



pav. 28 Atskiros gijos ryšio užmezgėjui ir gijos su įvykio išskyrėju užklausų apdorotojams modelio sekos diagrama.

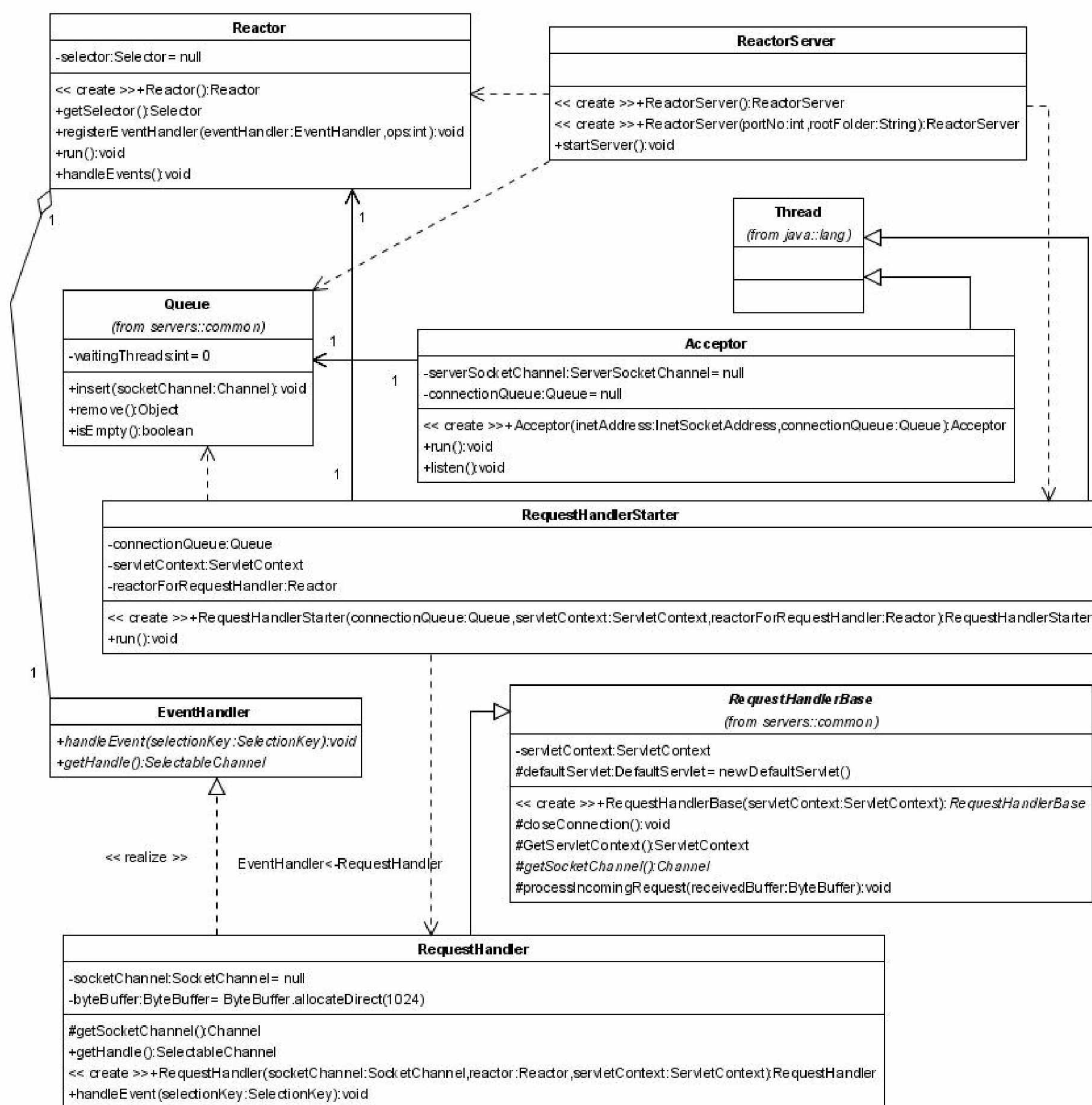
- Veiksmas prasideda nuo serverio paleidimo. Pirmajame žingsnyje sukuriamas yra *ReactorServer* objektas, kuris valdo pagrindinė programos eigą.
- Antrame žingsnyje yra sukuriamas reaktoriaus objektas (R*eactor*) skirtas užklausų apdorotojams. Kuriant objektą parametru yra nurodomas nepertraukiamas laukimo laikas milisekundėmis.
- Trečiame žingsnyje yra sukuriamas ryšio užmezgėjo objektas (*Acceptor*). Kuriant objektą parametrais yra perduodami prisijungimo adreso, informacijos konteksto ir reaktoriaus objektai
- Ketvirtame žingsnyje kviečiamas yra ryšio užmezgėjo metodas *listen*, kuris sukūrus savarankišką giją startuoja amžiną ciklą. Kiekvienos iteracijos metu sinchroniškai yra atliekami tokie veiksmai:
    - Pradedama laukti naujų ryšių užmezgimo užklausų. Laukimo stadijoje gija užsiblokuoja kol neatsiras naujų ryšio užmezgimo užklausų.
    - Sulaukus ryšio užmezgimo užklausos yra sukuriamas užklausų apdorotojo objektas (*Requesthandler*). Kuriant objektą parametrais yra perduodami prisijungimo, reaktoriaus bei informacijos konteksto objektai.
    - Sukūrus užklausos apdorotojo objektą užregistruojame jo suinteresuotumą skaitymo/rašymo įvykiais reaktoriaus objekte esančiame įvykio išskyrėjuje.

- Penktame žingsnyje kviečiamas yra užklausų apdorotojams skirto reaktoriaus metodas *handleEvents*, kuris sukūrus savarankišką gija startuoja amžiną ciklą. Kiekvienos iteracijos metu sinchroniškai yra atliekami tokie veiksmai:
    - Pagal nustatytą nepertraukiamo laukimo laiką įvykio išskyrėjas pradedama tikrinti ar neatsirado skaitymo/rašymo įvykių. Aptikus įvykį arba pasibaigus laukimo laikui tikrinimas yra nutraukiamas ir pereinama prie kito žingsnio.
    - Atsiradus skaitymo/rašymo įvykiui užklausų apdorotojas atitinkamai sureaguoja ir sinchroniškai nesiblokuojant apdoroja užklausą.

2.2.4. Atskiros gijos ryšio užmezgėjui, gijos prisijungimų eilės apdorojimui ir gijos su įvykio išskyrėju užklausų apdorotojams modelis.

Šito modelio skirtumą nuo praeitame skyriuje aprašyto sudaro tai, kad yra atskirtas ryšio užmezgėjas ir užklausų apdorotojai panaudojant bendrą prisijungimo eilę. Teoriškai šis sprendimas turi užtikrinti greitesnį ryšio užmezgimą sumažinant atmetamų prisijungimų skaičių. Taip yra manoma todėl, kad ryšio užmezgėjui nereikia laukti kol jis užregistruos naujai sukurtą užklausos apdorotoją įvykių išskyrėjo objekte. Minėtas laukimo laikas susideda iš to per kiek laiko bus nutrauktas įvykių atsiradimo tikrinimas. Todėl sukuriant atskirą užklausų apdorotojų kūrėjo objektą panaudojant bendrą prisijungimo eilė tikimasi išvengti perteklinio laukimo. Kadangi įvykių išskyrėjo objekte bus registruojami įvykių apdorotojai iš skirtingos gijos reikalingas tampa įvykių išskyrėjo parametrizavimas nepertraukiamu laukimo laiku.

Realizuojamo modelio struktūra yra nurodyta paveiksle pav. 32.



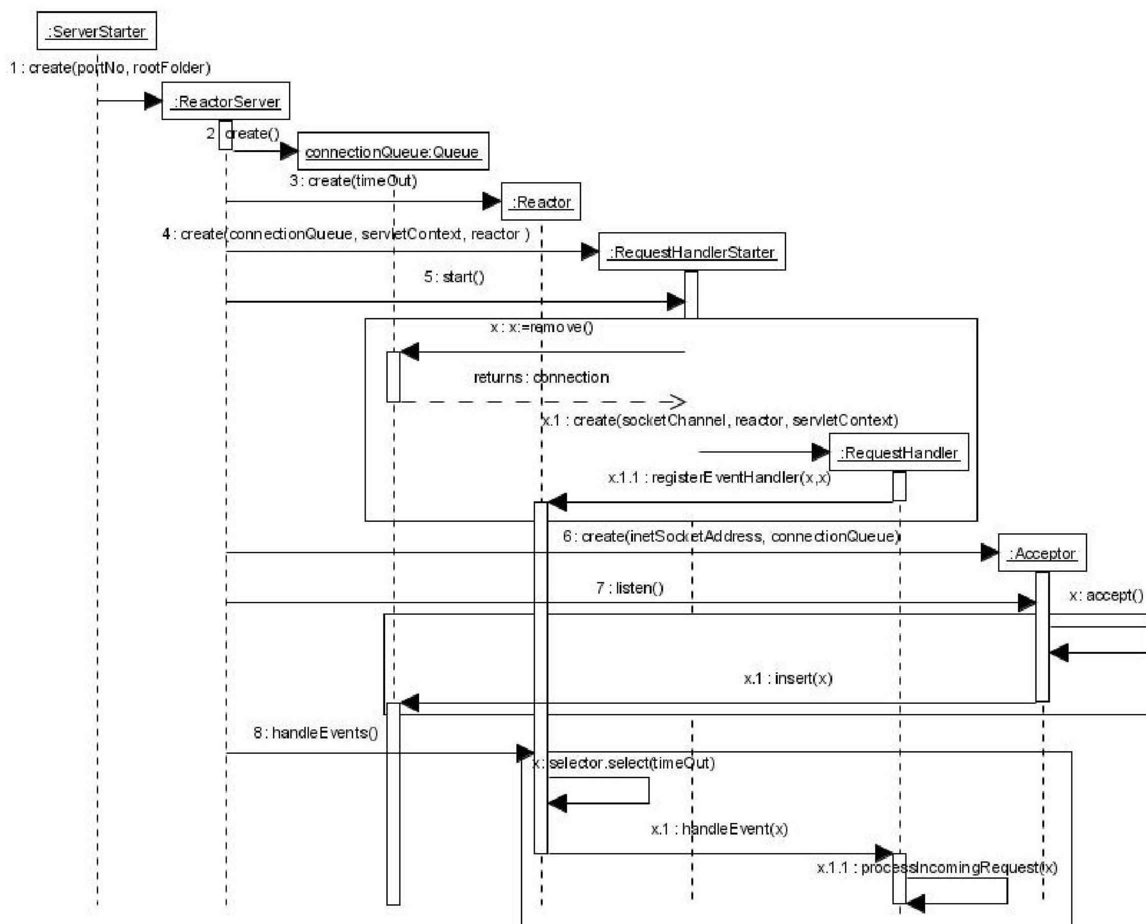
pav. 29 Atskiros gijos ryšio užmezgėjui, gijos prisijungimų eilės apdorojimui ir gijos su įvykio išskyrėju užklausų apdorotojams modelio klasių diagrama

Klasių aprašymai:

1. *ReactorServer* – pagrindinė modelio klasė atsakinga už prisijungimo eilės, reaktoriaus, užklausų apdorotojų kurėjo bei ryšio užmezgėjo objektų sukūrimą bei atitinkamų objektų darbo paleidimą.
2. *Queue* – prisijungimų eilė. Klasė atsakinga už prisijungimų kaupimą.
3. *Reactor* – reaktorius. Klasė atsakinga už įvykių išskyrėjo darbą bei už informavimą atitinkamų įvykių apdorotojų.
4. *EventHandler* – įvykio apdorotojas. Bazinė klasė, iš kurios yra paveldėtas skaitymo/rašymo įvykių apdorotojai.
5. *Acceptor* – ryšio užmezgėjas. Klasė atsakinga už ryšio užmezgimą bei prisijungimų pridėjimą prie bendros prisijungimo eilės..

6. *RequestHandlerStarter* – užklausų apdorotojų sukurėjas. Klasė atsakinga už užklausų apdorotojų sukūrimą naudojant bendrą prisijungimo eilę.
7. *RequestHandler* – užklausų apdorotojas. Klasė reaguojanti į skaitymo/rašymo įvykius. Atsakinga už užklausų apdorojimą.
8. *RequestHandlerBase* – bendra visoms konkurentiškumo valdymo modeliams užklausos vykdymo klasė, iš kurios yra paveldėtas užklausų apdorotojas.

Modelio veikimas yra parodytas sekos diagramoje pav. 33.



pav. 30 Atskiros gijos ryšio užmezgėjui, gijos prisijungimų eilės apdorojimui ir gijos su įvykio išskyrėju užklausų apdorotojams modelio sekos diagrama

- Veiksmas prasideda nuo serverio paleidimo. Pirmajame žingsnyje sukuriamas yra *ReactorServer* objektas, kuris valdo pagrindinė programos eigą.
- Antrame žingsnyje yra sukuriama tuščia bendra prisijungimo eile (*coonectionQueue*).
- Trečiame žingsnyje yra sukuriamas reaktoriaus objektas (R*eactor*) skirtas užklausų apdorotojams. Kuriant objektą parametru yra nurodomas nepertraukiamas laukimo laikas milisekundėmis.
- Ketvirtame žingsnyje yra sukuriamas užklausų apdorotojų kūrėjo objektas (*RequestHandlerStarter*). Kuriant objektą parametrais yra perduodami bendros prisijungimo eilės, informacijos konteksto bei reaktoriaus objektai.

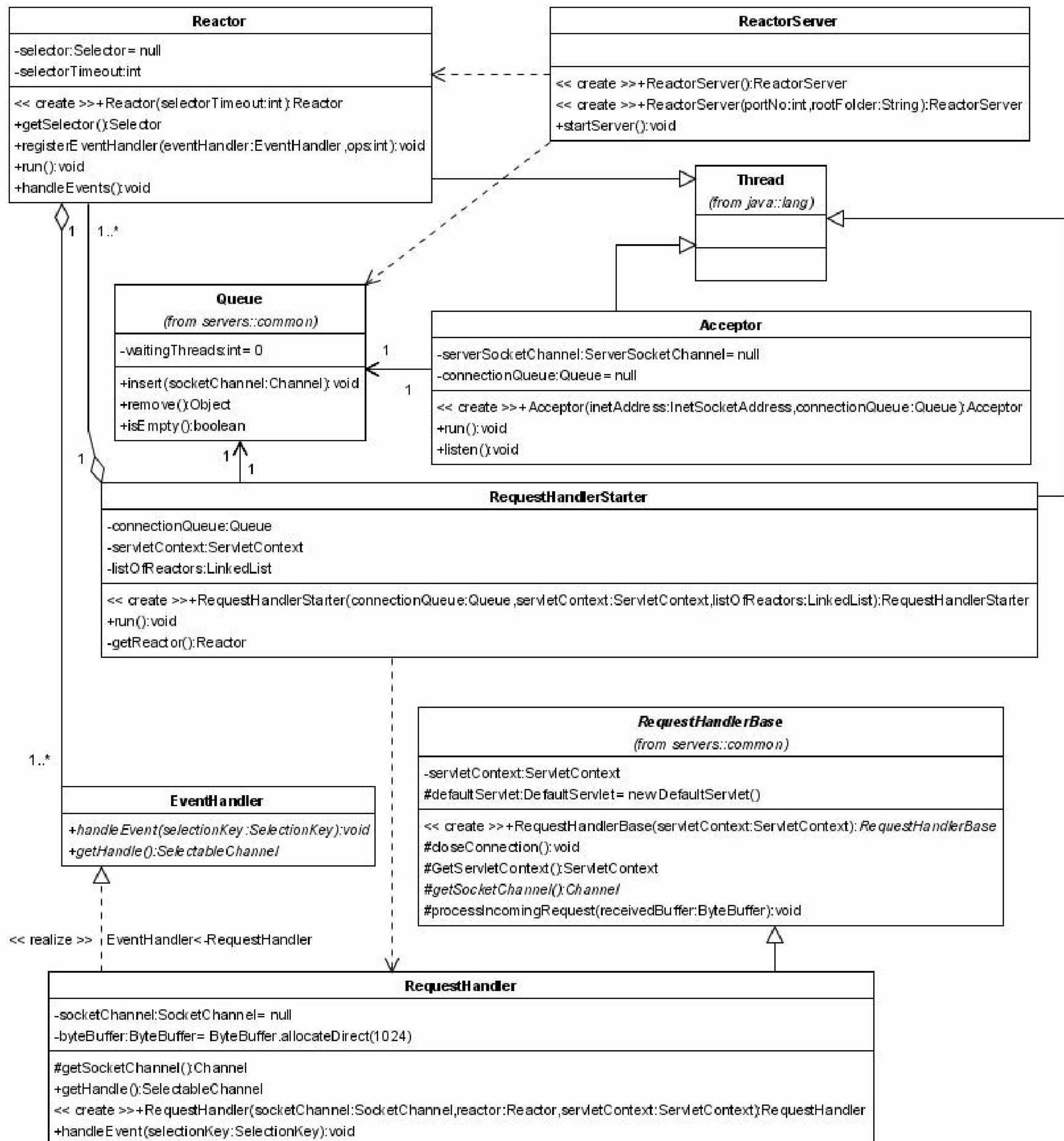
- Penktame žingsnyje kviečiamas yra užklausų apdorotojų kūrėjo metodas *start*, kuris sukūrus savarankišką giją startuoja amžiną ciklą. Kiekvienos iteracijos metu sinchroniškai yra atliekami tokie veiksmai:
  - Pradedama laukti naujų prisijungimų bendroje prisijungimų eilėje. Laukimo stadijoje gija užsiblokuoja kol prisijungimų eilė nebus papildyta.
  - Sulauks eilėje naujo prisijungimo yra sukuriamas užklausos apdorotojo objektas (*RequestHandler*). Kuriant objektą parametrais yra perduodami prisijungimo, reaktoriaus bei informacijos konteksto objektai.
  - Sukūrus užklausos apdorotojo objektą užregistruojame jo suinteresuotumą skaitymo/rašymo įvykiais reaktoriaus objekte esančiame įvykio išskyrėjuje.
- Šeštame žingsnyje yra sukuriamas ryšio užmezgėjo objektas (*Acceptor*). Kuriant objektą parametrais yra perduodami prisijungimo adreso bei bendros prisijungimo eilės objektai.
- Septintame žingsnyje kviečiamas yra ryšio užmezgėjo metodas *listen*, kuris sukūrus savarankišką giją startuoja amžiną ciklą. Kiekvienos iteracijos metu sinchroniškai yra atliekami tokie veiksmai:
  - Pradedama laukti naujų ryšių užmezgimo užklausų. Laukimo stadijoje gija užsiblokuoja kol neatsiras naujų ryšio užmezgimo užklausų.
  - Sulaukus ryšio užmezgimo užklausą yra užmegztas prisijungimas yra pridedamas prie bendros prisijungimų eilės.
- Aštuntame žingsnyje kviečiamas yra užklausų apdorotojams skirto reaktoriaus metodas *handleEvents*, kuris sukūrus savarankišką gija startuoja amžiną ciklą. Kiekvienos iteracijos metu sinchroniškai yra atliekami tokie veiksmai:
  - Pagal nustatytą nepertraukiamo laukimo laiką įvykio išskyrėjas pradedama tikrinti ar neatsirado skaitymo/rašymo įvykių. Aptikus įvykį arba pasibaigus laukimo laikui tikrinimas yra nutraukiamas ir pereinama prie kito žingsnio.
  - Atsiradus skaitymo/rašymo įvykiui užklausų apdorotojas atitinkamai sureaguoja ir sinchroniškai nesiblokuojant apdoroja užklausą.

2.2.5. Atskiros gijos ryšio užmezgėjui, gijos prisijungimo eilės apdorojimui ir parametrizuotas gijų skaičius su savais įvykių išskyrėjais užklausų apdorotojams modelis.

Šio modelio skirtumas nuo praeitame skyriuje aprašyto modelio sudaro parametrizuotas gijų skaičius. Konkurentiškumą valdantis komponentas yra parametrizuojamas darbinių gijų skaičiumi su savais įvykių išskyrėjais. Tokio modelio atsiradimas sąlygojo tas dalykas, kad esant vienam įvykių išskyrėjui visos užklausų apdorojimo operacijos yra vykdomos sinchroniškai viena po kitos.

Todėl sukūrus daugiau gijų su įvykių išskyrėjais teoriškai yra tikimasi, kad lygiagretūs užklausų apdorojimas gali pagreitinti bendrą užklausų apdorojimo darbą. Gijos yra sukuriamos startuojant serverio darbui ir prieinamos užklausų apdorotojų kūrėjui su vienodu prioritetu. Kadangi užklausų apdorotojų kūrėjas registruos įvykių apdorotojus įvykių išskyrėjuose iš skirtingos gijos reikalingas tampa įvykių išskyrėjų parametrizavimas nepertraukiamu laukimo laiku.

Realizuojamo modelio struktūra yra nurodyta paveiksle pav. 34.



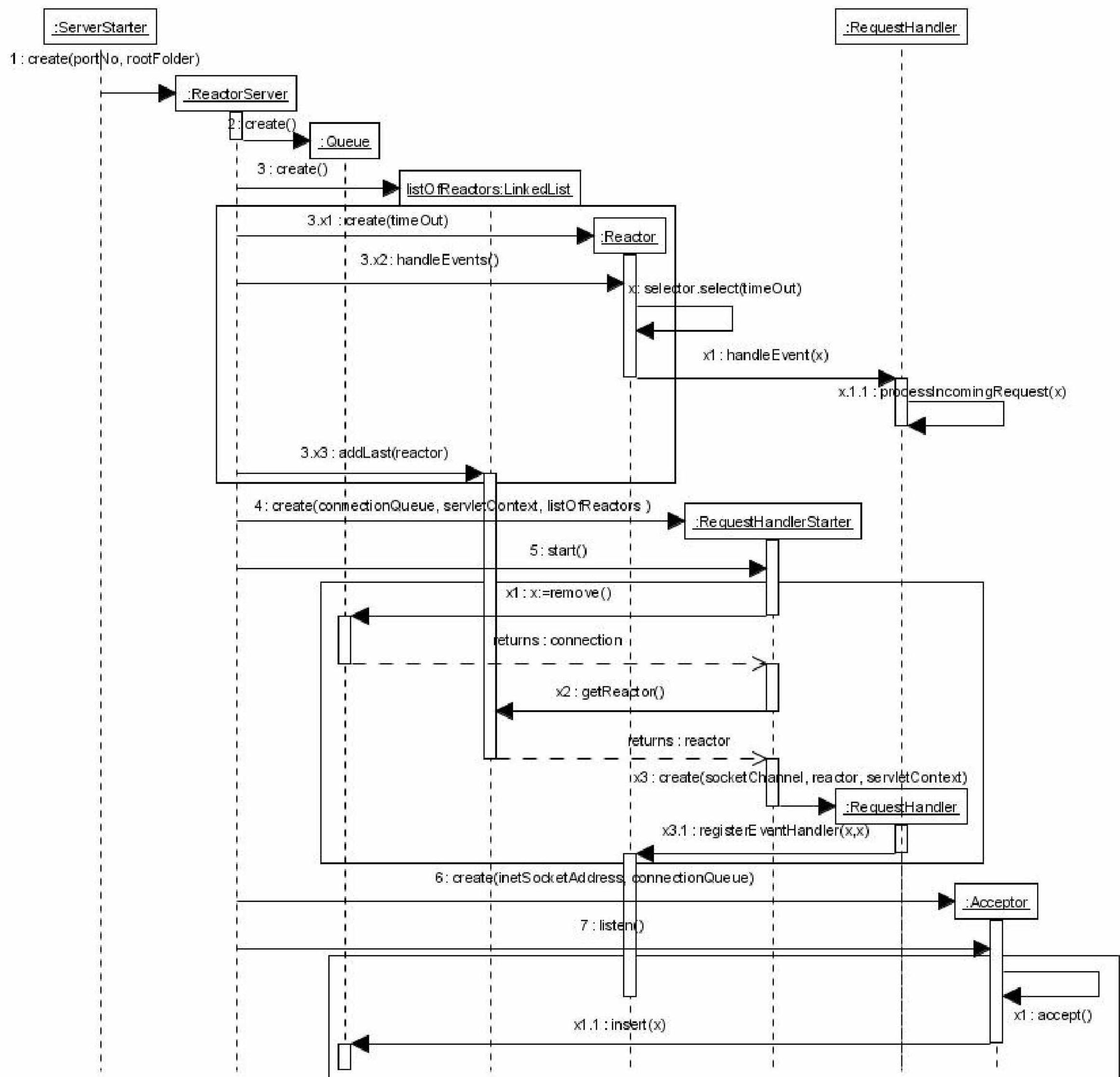
pav. 31 Atskiros gijos ryšio užmezgėjui, gijos prisijungimo eilės apdorojimui ir parametrizuotas gijų skaičius su savais įvykių išskyrėjais užklausų apdorotojams modelio klasės diagrama.

Klasių aprašymai:

1. *ReactorServer* – pagrindinė modelio klasė atsakinga už prisijungimo eilės, reaktorių, užklausų apdorotojų kurėjo bei ryšio užmezgėjo objektų sukūrimą bei atitinkamų objektų darbo paleidimą.
2. *Queue* – prisijungimų eilė. Klasė atsakinga už prisijungimų kaupimą.

3. *Reactor* – reaktorius. Klasė atsakinga už įvykių išskyrėjo darbą bei už informavimą atitinkamų įvykių apdorotojų.
4. *EventHandler* – įvykio apdorotojas. Bazinė klasė, iš kurios yra paveldėtas skaitymo/rašymo įvykių apdorotojai.
5. *Acceptor* – ryšio užmezgėjas. Klasė atsakinga už ryšio užmezgimą bei prisijungimų pridėjimą prie bendros prisijungimo eilės..
6. *RequestHandlerStarter* – užklausų apdorotojų sukurėjas. Klasė atsakinga už užklausų apdorotojų objektų sukūrimą naudojant bendrą prisijungimo eilę ir bei tų objektų užregistravimą tam tikrame reaktoriuje.
7. *RequestHandler* – užklausų apdorotojas. Klasė reaguojanti į skaitymo/rašymo įvykius. Atsakinga už užklausų apdorojimą.
8. *RequestHandlerBase* – bendra visoms konkurentiškumo valdymo modeliams užklausos vykdymo klasė, iš kurios yra paveldėtas užklausų apdorotojas.

Modelio veikimas yra parodytas sekos diagramoje pav. 35.



pav. 32 Atskiros gijos ryšio užmezgėjui, gijos prisijungimo eilės apdorojimui ir parametrizuotas gijų skaičius su savais įvykių išskyrėjais užklausų apdorotojams modelio sekos diagrama.

- Veiksmas prasideda nuo serverio paleidimo. Pirmajame žingsnyje sukuriamas yra *ReactorServer* objektas, kuris valdo pagrindinė programos eigą. Kuriant objektą parametrais yra perduodami porto numeris, informacijos katalogas ir darbinių gijų skaičius.
- Antrame žingsnyje yra sukuriama tuščia bendra prisijungimo eile (*coonectionQueue*).
- Trečiame žingsnyje yra sukuriama tuščia reaktorių grupė (*listOfReactors*).
- Sekančiame žingsnyje startuoja ciklą, kurio iteracijų skaičius lygus darbinių gijų skaičiui. Kiekvienos iteracijos metu yra atliekami tokie veiksmai:
    - o Sukuriamas reaktoriaus objektas (R*eactor*) skirtas užklausų apdorotojams. Kuriant objektą parametru yra nurodomas nepertraukiamas laukimo laikas milisekundėmis.

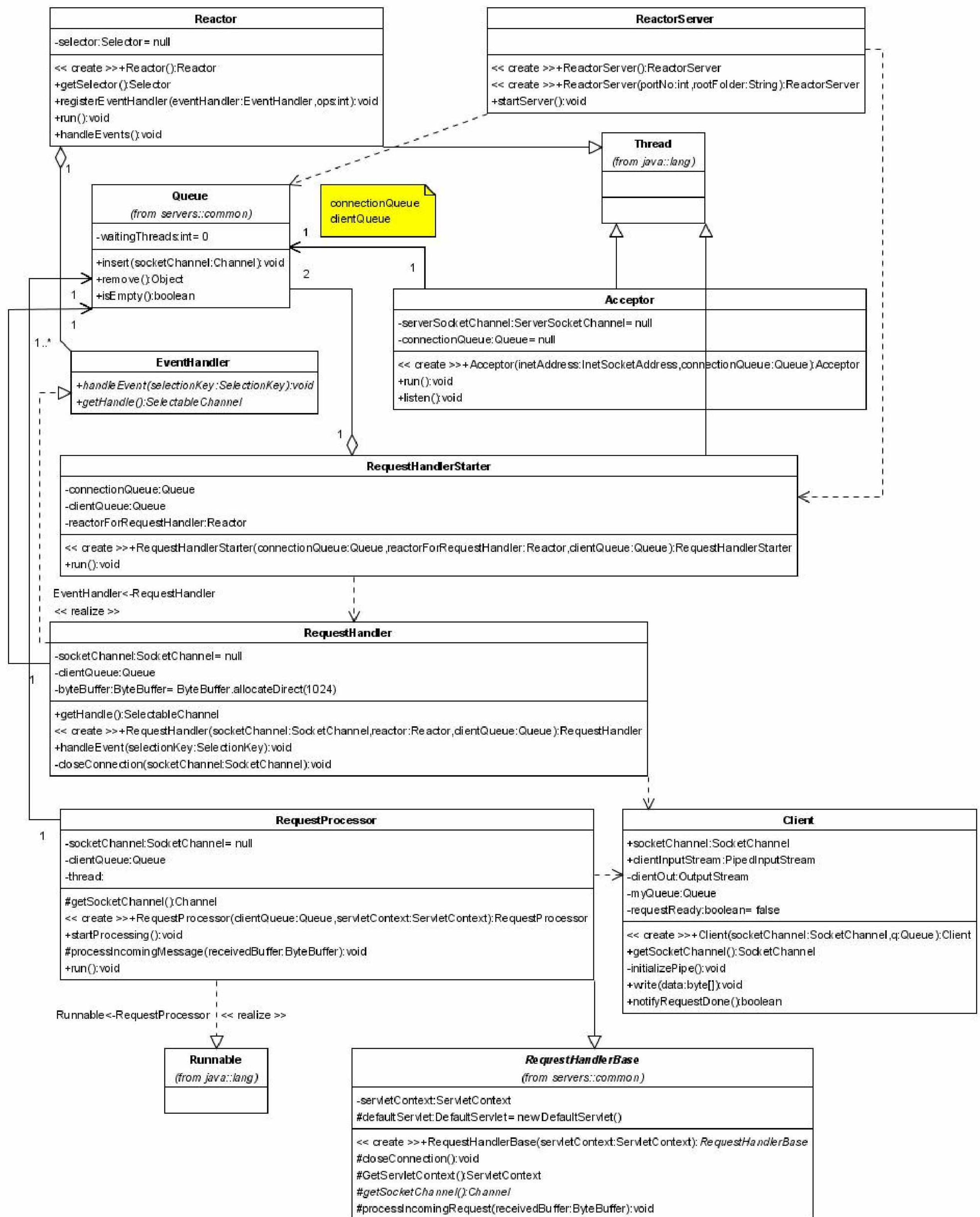
- Sekančiame žingsnyje kviečiamas yra užklausų apdorotojams skirto reaktoriaus metodas *handleEvents*, kuris sukūrus savarankišką gija startuoja amžiną ciklą. Kiekvienos iteracijos metu sinchroniškai yra atliekami tokie veiksmai:
  - Pagal nustatytą nepertraukiamo laukimo laiką įvykio išskyrėjas pradedama tikrinti ar neatsirado skaitymo/rašymo įvykių. Aptikus įvykį arba pasibaigus laukimo laikui tikrinimas yra nutraukiamas ir pereinama prie kito žingsnio.
  - Atsiradus skaitymo/rašymo įvykiui užklausų apdorotojas atitinkamai sureaguoja ir sinchroniškai nesiblokuojant apdoroja užklausą.
- Sukurtas užklausų apdorotojo objektas pridedamas prie reaktorių grupės.

- Ketvirtame žingsnyje yra sukuriamas užklausų apdorotojų kūrėjo objektas (*RequestHandlerStarter*). Kuriant objektą parametrais yra perduodami bendros prisijungimo eilės, informacijos konteksto bei reaktorių grupės objektai.
- Penktame žingsnyje kviečiamas yra užklausų apdorotojų kūrėjo metodas *start*, kuris sukūrus savarankišką giją startuoja amžiną ciklą. Kiekvienos iteracijos metu sinchroniškai yra atliekami tokie veiksmai:
  - Pradedama laukti naujų prisijungimų bendroje prisijungimų eilėje. Laukimo stadijoje gija užsiblokuoja kol prisijungimų eilė nebus papildyta.
  - Sulauks eilėje naujo prisijungimo yra sukuriamas užklausos apdorotojo objektas (*RequestHandler*). Kuriant objektą parametrais yra perduodami prisijungimo, reaktoriaus bei informacijos konteksto objektai. Perduodamas parametru reaktorius yra tam tikru momentu mažiausiai turintis užregistruotų įvykių apdorotojų reaktorius. Tokiu būdu bandoma yra užtikrinti vienodą užregistruotų skirtingose įvykių išskyrėjuose įvykių apdorotojų paskirtymą.
  - Sukūrus užklausos apdorotojo objektą užregistruojame jo suinteresuotumą skaitymo/rašymo įvykiais reaktoriaus objekte esančiame įvykio išskyrėjuje.
- Šeštame žingsnyje yra sukuriamas ryšio užmezgėjo objektas (*Acceptor*). Kuriant objektą parametrais yra perduodami prisijungimo adreso bei bendros prisijungimo eilės objektai.
- Septintame žingsnyje kviečiamas yra ryšio užmezgėjo metodas *listen*, kuris sukūrus savarankišką giją startuoja amžiną ciklą. Kiekvienos iteracijos metu sinchroniškai yra atliekami tokie veiksmai:
  - Pradedama laukti naujų ryšių užmezgimo užklausų. Laukimo stadijoje gija užsiblokuoja kol neatsiras naujų ryšio užmezgimo užklausų.
  - Sulaukus ryšio užmezgimo užklausą yra užmegztas prisijungimas yra pridedamas prie bendros prisijungimų eilės.

2.2.6. Atskiros gijos ryšio užmezgėjui, gijos prisijungimo eilės apdorojimui, gijos su įvykių išskyrėju užklausų apdorotojams bei parametrizuotas gijų skaičius skirtu užklausų vykdytojams modelis.

Šis modelis yra aprašyto 2.2.4 skyriuje atskiros gijos ryšio užmezgėjui be įvykio išskyrėjo, gijos prisijungimo eilės apdorojimui ir gijos su įvykio išskyrėju užklausų apdorotojams modelio modifikacija. Skirtumą sudaro darbinių gijų skirtu užklausų vykdytojams buvimas. Tokio modelio atsiradimą sąlygojo tas dalykas, kad esant vienam įvykių išskyrėjui visos užklausų apdorojimo operacijos yra vykdomos sinchroniškai viena po kitos. Todėl sukūrus gijų grupę, kuriai bus deleguotas užklausų vykdymas teoriškai turėtų sumažėti bendras užklausų apdorojimo laikas. Taip yra manoma todėl, kad įvykių išskyrėjas delegavęs užklausos vykdymą darbinei gijai, gali vėl pradėti laukti įvykių nežiūrint į tai kiek ilgai bus vykdoma užklausą. Kadangi užklausų apdorotojų kūrėjas registruos įvykių apdorotojus įvykių išskyrėjuje, kuris yra vykdomas skirtingoje gijoje reikalingas tampa įvykių išskyrėjo parametrizavimas nepertraukiamu laukimo laiku.

Realizuojamo modelio struktūra yra nurodyta paveiksle pav. 33.



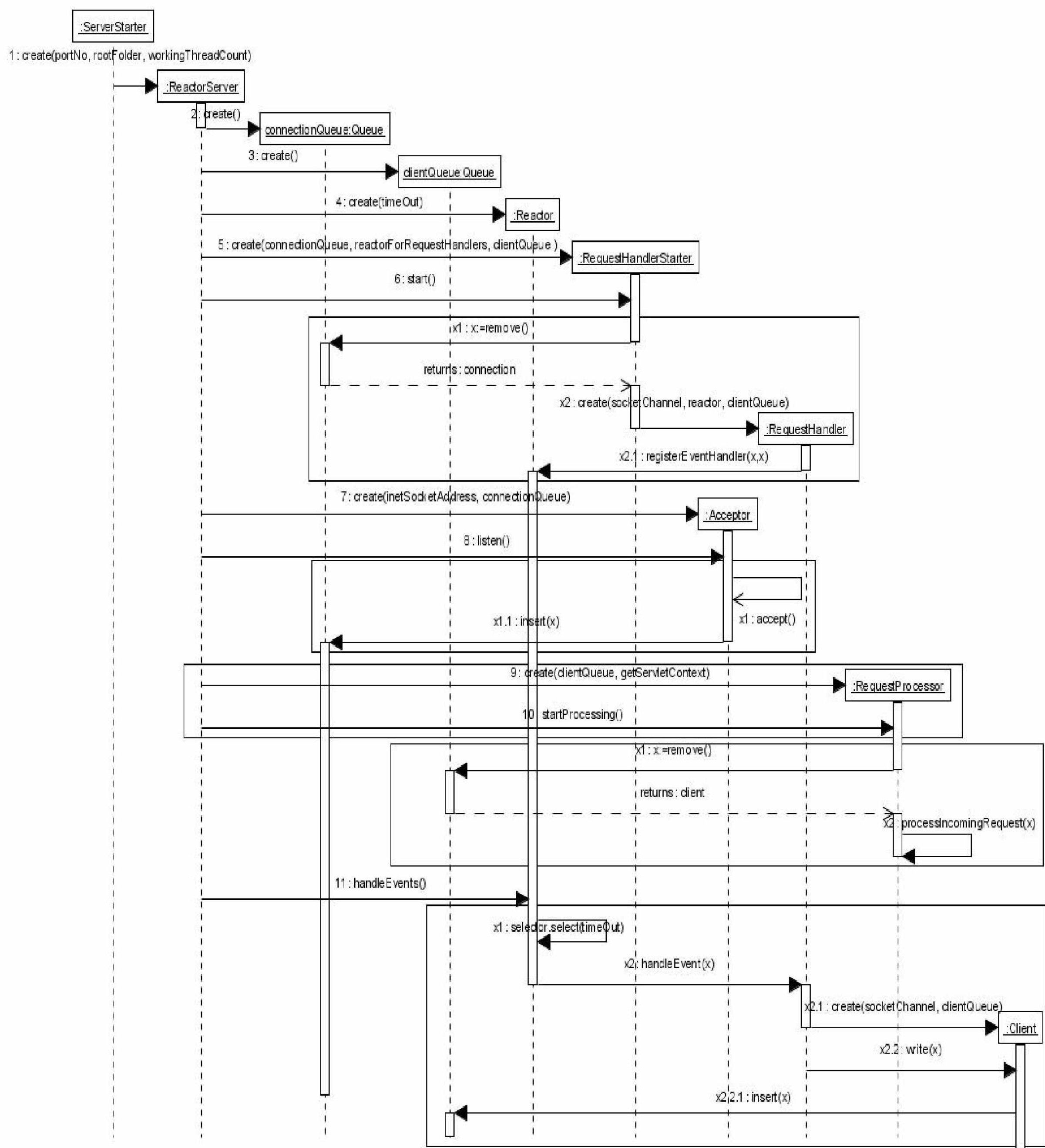
pav. 33 Atskiros gijos ryšio užmezgėjui, gijos prisijungimo eilės apdorojimui, gijos su įvykių išskyrėju užklausų apdorotojams bei parametrizuotas gijų skaičius skirtu užklausų vykdytojams modelio klasių diagrama

Klasių aprašymai:

1. *ReactorServer* – pagrindinė modelio klasė atsakinga už prisijungimo eilės, klientų eilės, reaktoriaus, užklausų apdorotojų kūrėjo, ryšio užmezgėjo objektų bei darbinių gijų skirtu užklausų vykdytojams sukūrimą bei atitinkamų objektų darbo paleidimą.

2. *Queue* – prisijungimų/klientų eilė. Klasė atsakinga už prisijungimų/klientų kaupimą.
3. *Reactor* – reaktorius. Klasė atsakinga už įvykių išskyrėjo darbą bei už informavimą atitinkamų įvykių apdorotojų.
4. *EventHandler* – įvykio apdorotojas. Bazinė klasė, iš kurios yra paveldėtas skaitymo/rašymo įvykių apdorotojai.
5. *Acceptor* – ryšio užmezgėjas. Klasė atsakinga už ryšio užmezgimą bei prisijungimų pridėjimą prie bendros prisijungimo eilės..
6. *RequestHandlerStarter* – užklausų apdorotojų sukurėjas. Klasė atsakinga už užklausų apdorotojų sukūrimą naudojant bendrą prisijungimo eilę.
7. *RequestHandler* – užklausų apdorotojas. Klasė reaguojanti į skaitymo/rašymo įvykius. Atsakinga už užklausų apdorojimą.
8. *Client* – kliento užklausa. Klasė atsakinga už užklausos laikymą.
9. *RequestProcessor* – užklausos vykdytojas. Klasė atsakinga už užklausos vykdymą.
10. *RequestHandlerBase* – bendra visoms konkurentiškumo valdymo modeliams užklausos vykdymo klasė, iš kurios yra paveldėtas užklausos vykdytojas.

Modelio veikimas yra parodytas sekos diagramoje pav. 34.



pav. 34 Atskiros gijos ryšio užmezgėjui, gijos prisijungimo eilės apdorojimui, gijos su įvykių išskyrėju užklausų apdorotojams bei parametrizuotas gijų skaičius skirtu užklausų vykdytojams modelio sekosdiagrama.

- Veiksmas prasideda nuo serverio paleidimo. Pirmajame žingsnyje sukuriamas yra *ReactorServer* objektas, kuris valdo pagrindinė programos eigą. Kuriant objektą parametrais yra perduodami porto numeris, informacijos katalogas ir darbinių gijų skaičius.
- Antrame žingsnyje yra sukuriama tuščia bendra prisijungimo eiėe (*connectionQueue*).
- Trečiame žingsnyje yra sukuriama tuščia klientų užklausų eilė (*clientQueue*).
- Ketvirtame žingsnyje yra sukuriamas reaktoriaus objektas (R*eactor*) skirtas užklausų apdorotojams. Kuriant objektą parametru yra nurodomas nepertraukiamas laukimo laikas milisekundėmis.

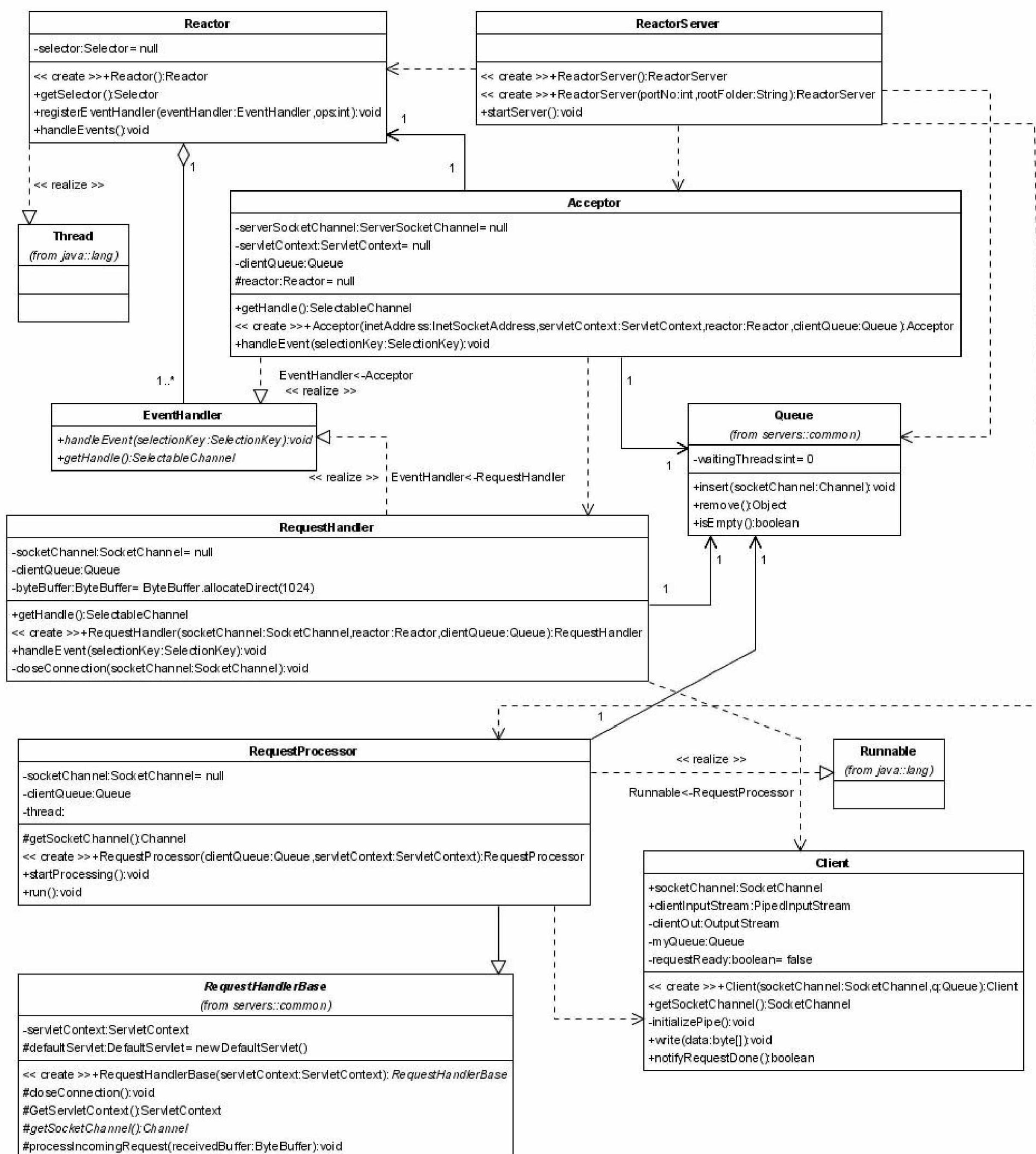
- Penktame žingsnyje yra sukuriamas užklausų apdorotojų kūrėjo objektas (*RequestHandlerStarter*). Kuriant objektą parametrais yra perduodami bendros prisijungimo eilės, reaktoriaus bei klientų eilės objektai.
- Šeštame žingsnyje kviečiamas yra užklausų apdorotojų kūrėjo metodas *start*, kuris sukūrus savarankišką giją startuoja amžiną ciklą. Kiekvienos iteracijos metu sinchroniškai yra atliekami tokie veiksmai:
  - Pradedama laukti naujų prisijungimų bendroje prisijungimų eilėje. Laukimo stadijoje gija užsiblokuoja kol prisijungimų eilė nebus papildyta.
  - Sulaukus eilėje naujo prisijungimo yra sukuriamas užklausos apdorotojo objektas (*RequestHandler*). Kuriant objektą parametrais yra perduodami prisijungimo, reaktoriaus bei klientų užklausų eilės objektai.
  - Sukūrus užklausos apdorotojo objektą užregistruojame jo suinteresuotumą skaitymo/rašymo įvykiais reaktoriaus objekte esančiame įvykio išskyrėjuje.
- Septintame žingsnyje yra sukuriamas ryšio užmezgėjo objektas (*Acceptor*). Kuriant objektą parametrais yra perduodami prisijungimo adreso bei bendros prisijungimo eilės objektai.
- Aštuntame žingsnyje kviečiamas yra ryšio užmezgėjo metodas *listen*, kuris sukūrus savarankišką giją startuoja amžiną ciklą. Kiekvienos iteracijos metu sinchroniškai yra atliekami tokie veiksmai:
  - Pradedama laukti naujų ryšių užmezgimo užklausų. Laukimo stadijoje gija užsiblokuoja kol neatsiras naujų ryšio užmezgimo užklausų.
  - Sulaukus ryšio užmezgimo užklausą užmegztas prisijungimas yra pridedamas prie bendros prisijungimų eilės.
- Sekančiame žingsnyje yra startuojamas ciklas, kurio iteracijų skaičius lygus darbinių gijų skaičiui. Kiekvienos iteracijos metu yra atliekami tokie veiksmai:
  - Sukuriamas yra užklausos vykdytojo objektas (*RequestProcessor*). Kuriant objektą parametrais yra perduodami klientų užklausų eilės bei informacijos konteksto objektai.
  - Kviečiamas yra užklausų vykdytojo metodas *startProcessing*, kuris sukūrus savarankišką giją startuoja amžiną ciklą. Kiekvienos iteracijos metu sinchroniškai yra atliekami tokie veiksmai:
    - Pradedama laukti klientų užklausų eilėje. Laukimo stadijoje gija užsiblokuoja kol klientų užklausų eilė nebus papildyta.
    - Sulaukus eilėje naujos kliento užklausos pradedamas jos sinchroniškas ir nesiblokuojantis vykdymas.

- Vienuoliktame žingsnyje kviečiamas yra užklausų apdorotojams skirto reaktoriaus metodas *handleEvents*, kuris sukūrus savarankišką gija startuoja amžiną ciklą. Kiekvienos iteracijos metu sinchroniškai yra atliekami tokie veiksmai:
    - Pagal nustatytą nepertraukiamo laukimo laiką įvykio išskyrėjas pradedama tikrinti ar neatsirado skaitymo/rašymo įvykių. Aptikus įvykį arba pasibaigus laukimo laikui tikrinimas yra nutraukiamas ir pereinama prie kito žingsnio.
    - Atsiradus skaitymo/rašymo įvykiui užklausų apdorotojas atitinkamai sureaguoja ir atlieka tokius veiksmus:
        - Nuskaito užklausą
        - Sukuria kliento užklausos objektą (*Client*).
        - Kviečia kliento objekto metodą *write*, kuris užrašus užklausą prideda save kaip objektą prie klientų užklausos eilės.

2.2.7. Vienos gijos ir vieno įvykių išskyrėjo modelis su parametrizuota gijų grupe skirta užklausų vykdytojams.

Šio ir aprašyto praeitame skyriuje modelių principai yra labai panašūs. Skirtumas yra tas, kad aprašomajame modelyje nėra naudojama prisijungimo eilė, o ryšio užmezgėjo sąveiką su užklausos apdorotojais yra realizuojama taip kaip tas yra daroma vienos gijos ir vieno įvykių išskyrėjo modelyje. Kadangi užklausų vykdymas yra atskirtas nuo ryšio užmezgimo bei užklausų nuskaitymo, o taip pat nėra prisijungimo eilės aptarnavimo veiksmų teoriškai šis sprendimas turi būti pranašesnis už modelį aprašytą praeitame skyriuje laiko reikalingo įvykdyti užklausas atžvilgiu.

Realizuojamo modelio struktūra yra nurodyta paveiksle pav. 38.



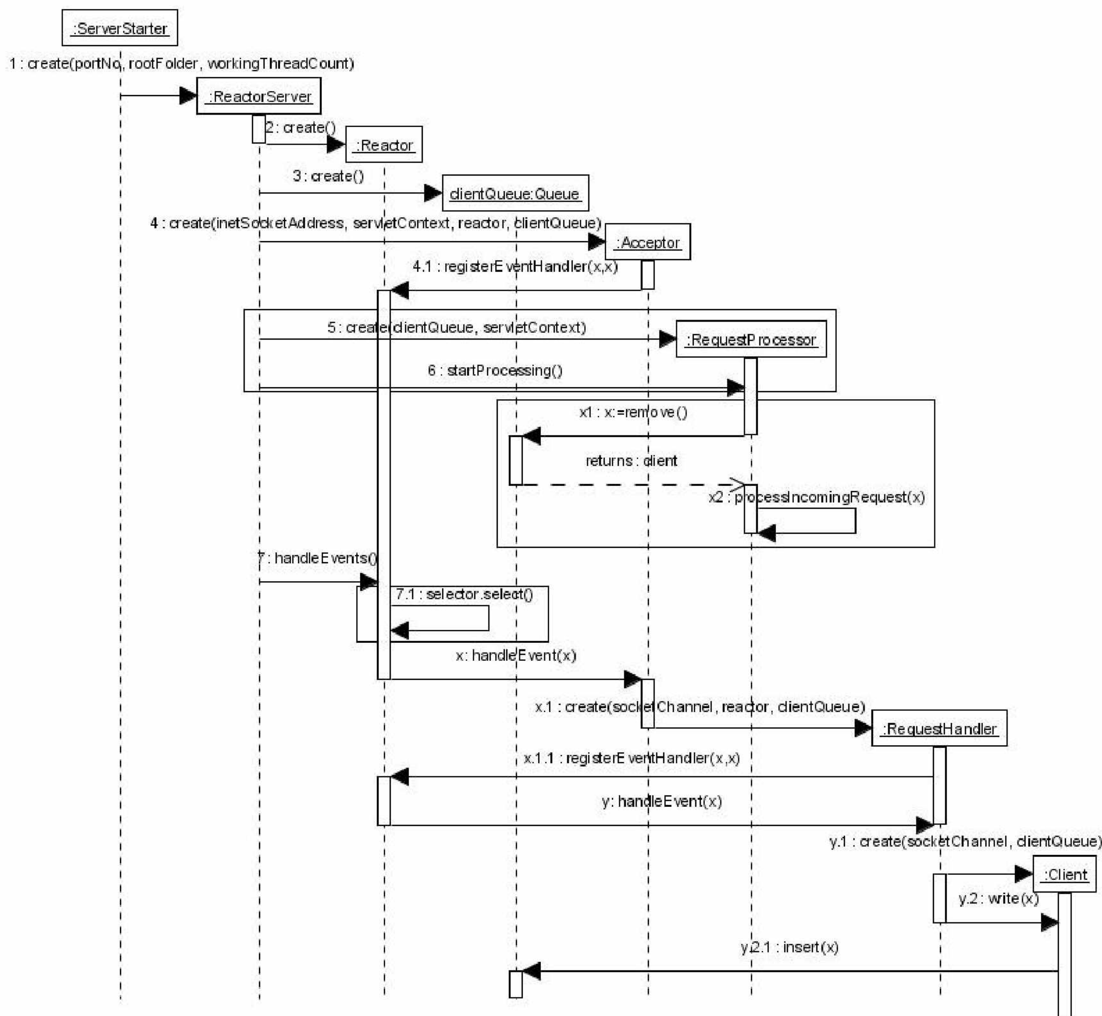
pav. 35 Vienos gijos ir vieno įvykių išskyrėjo modelio su parametrizuota gijų grupe skirta užklausų vykdytojams klasių diagrama.

Klasių aprašymai:

1. *ReactorServer* – pagrindinė modelio klasė atsakinga už klientų užklausų eilės, reaktoriaus, užklausų apdorotojų kūrėjo, ryšio užmezgėjo objektų bei darbinių gijų skirtu užklausų vykdytojams sukūrimą bei atitinkamų objektų darbo paleidimą.
2. *Queue* – klientų užklausų eilė. Klasė atsakinga už klientų užklausų kaupimą.
3. *Reactor* – reaktorius. Klasė atsakinga už įvykių išskyrėjo darbą bei už informavimą atitinkamų įvykių apdorotojų.

4. *EventHandler* – įvykio apdorotojas. Bazinė klasė, iš kurios yra paveldėtas prisijungimų ir skaitymo/rašymo įvykių apdorotojai.
5. *Acceptor* – ryšio užmezgėjas. Klasė reaguojanti į prisijungimo įvykius. Atsakinga už ryšio užmezgimą bei prisijungimų bei už užklausų apdorotojų sukūrimą.
6. *RequestHandler* – užklausų apdorotojas. Klasė reaguojanti į skaitymo/rašymo įvykius. Atsakinga už užklausų apdorojimą.
7. *Client* – kliento užklausa. Klasė atsakinga už užklausos laikymą.
8. *RequestProcessor* – užklausos vykdytojas. Klasė atsakinga už užklausos vykdymą.
9. *RequestHandlerBase* – bendra visoms konkurentiškumo valdymo modeliams užklausos vykdymo klasė, iš kurios yra paveldėtas užklausos vykdytojas.

Modelio veikimas yra parodytas sekos diagramoje pav. 39.



pav. 36 Vienos gijos ir vieno įvykių išskyrėjo modelio su parametrizuota gijų grupe skirta užklausų vykdytojams sekos diagrama.

- Veiksmas prasideda nuo serverio paleidimo. Pirmajame žingsnyje sukuriamas yra *ReactorServer* objektas, kuris valdo pagrindinė programos eigą. Kuriant objektą parametrais yra perduodami porto numeris, informacijos katalogas ir darbinių gijų skaičius.

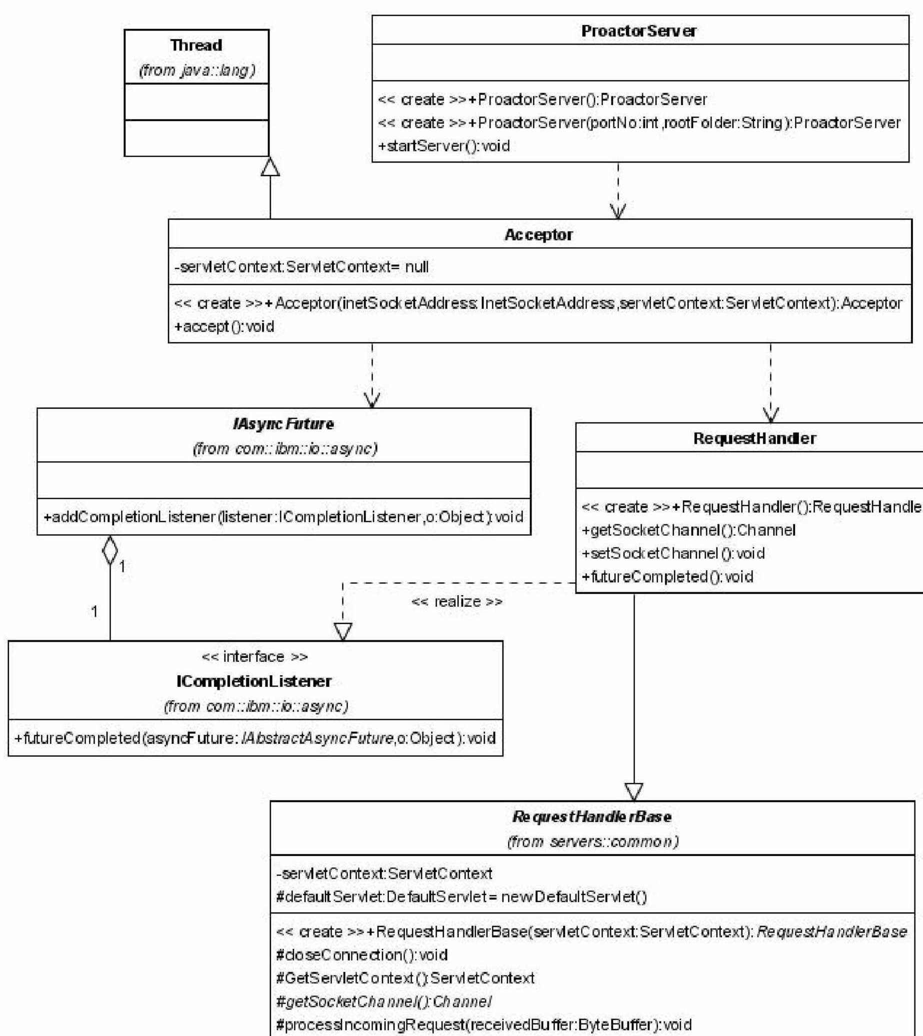
- Antrame žingsnyje yra sukuriamas reaktoriaus objektas (R*eactor*).
- Trečiame žingsnyje yra sukuriama tuščia klientų užklausų eilė (*clientQueue*).
- Ketvirtame žingsnyje yra sukuriamas ryšio užmezgėjo objektas (*Acceptor*).). Kuriant objektą parametrais yra perduodami prisijungimo adreso, informacijos konteksto, reaktoriaus bei klientų užklausų objektai.
    - Sukūrus ryšio užmezgėjo objektą, registruojame jo suinteresuotumą prisijungimo įvykiais reaktoriaus objekte esančiame įvykio išskyrėjuje.
- Sekančiame žingsnyje yra startuojamas ciklas, kurio iteracijų skaičius lygus yra darbinių gijų skaičiui. Kiekvienos iteracijos metu yra atliekami tokie veiksmai:
    - Sukuriamas yra užklausos vykdytojo objektas (*RequestProcessor*). Kuriant objektą parametrais yra perduodami klientų užklausų eilės bei informacijos konteksto objektai.
    - Kviečiamas yra užklausų vykdytojo metodas *startProcessing*, kuris sukūrus savarankišką giją startuoja amžiną ciklą. Kiekvienos iteracijos metu sinchroniškai yra atliekami tokie veiksmai:
        - Pradedama laukti klientų užklausų atsiradimo eilėje. Laukimo stadijoje gija užsiblokuoja kol klientų užklausų eilė nebus papildyta.
        - Sulaukus eilėje naujos kliento užklausos pradedamas jos sinchroniškas ir nesiblokuojantis vykdymas.
- Septintame žingsnyje kviečiamas yra reaktoriaus metodas *handleEvents*, kuris startuoja amžiną ciklą. Kiekvienos iteracijos metu sinchroniškai yra atliekami tokie veiksmai:
    - Įvykio išskyrėjas pradeda tikrinti ar neatsirado prisijungimo/skaitymo/rašymo įvykių. Aptikus įvykį tikrinimas yra nutraukiamas ir pereinama prie kito žingsnio.
    - Atsiradus prisijungimo įvykiui ryšio užmezgėjas atitinkamai sureaguoja ir sukuria užklausos apdorotojo objektą (*RequestHandler*). Kuriant objektą parametrais yra perduodami prisijungimo, reaktoriaus bei klientų užklausos eilės objektai.
        - Sukūrus užklausos apdorotojo objektą, registruojame jo suinteresuotumą skaitymo/rašymo įvykiais reaktoriaus objekte (R*eactor*) esančiame įvykio išskyrėjuje.
    - Atsiradus skaitymo/rašymo įvykiui užklausų apdorotojas atitinkamai sureaguoja ir atlieka tokius veiksmus:
        - Nuskaito užklausą
        - Sukuria kliento užklausos objektą (*Client*).
        - Kviečia kliento objekto metodą *write*, kuris užrašus užklausą prideda save kaip objektą prie klientų užklausos eilės.

2.3. Asinchroninis užklausų apdorojimas (Proaktoriaus schema)

2.3.1. Vienos gijos ir grįžtamojo kvietimo modelis.

Šitas modelis yra iš esmės aprašyto skyriuje 1.2.2 proaktoriaus schemos pagrindu valdomo konkurentiškumo modelio atkartojimas ir isskirtinių pakeitimų neturi. Aprašyta žemiau modelio struktūra ir veikimas sudaro bazę sekančioms modelio modifikacijoms. Modelio principas išliko tas pats, t.y. ryšių užmezgėjas sinchroniškai užmezginėja naujus prisijungimus, o sukurti užklausų apdorotojai užklausas vykdo asinchroniškai deleguodami skaitymo/rašymo veiksmus operacinėj sistemai, kuri atlikus patikėtą darbą iškviečia užregistruotą grįžtamąjį metodą.

Realizuojamo modelio struktūra yra nurodyta paveiksle pav. 37.



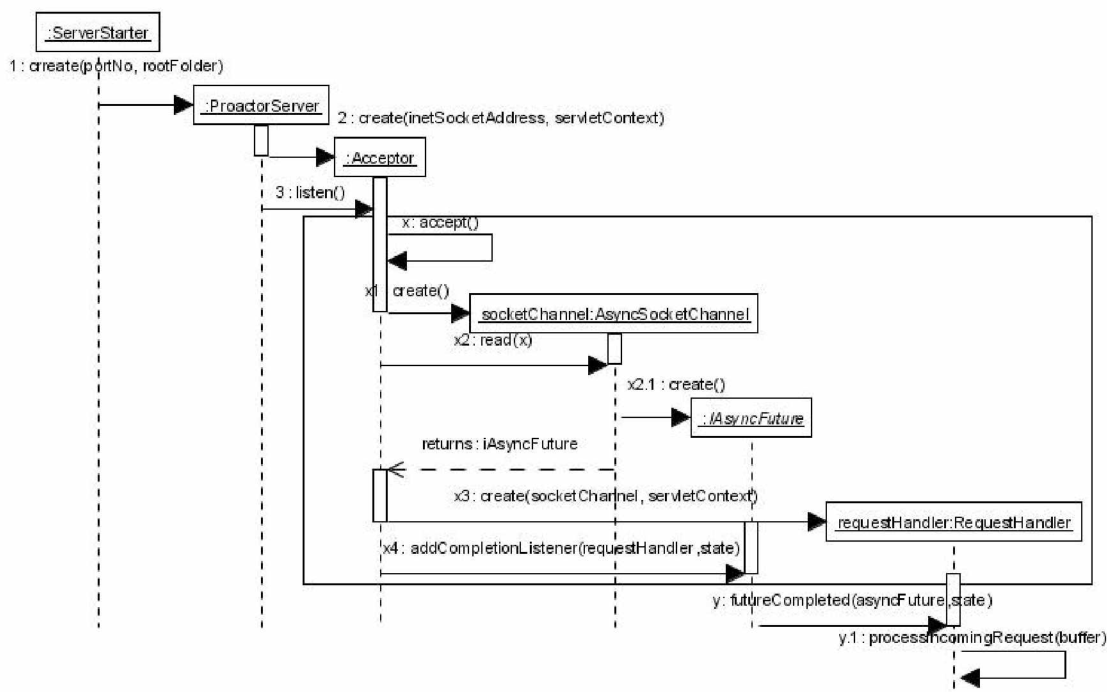
pav. 37 Vienos gijos ir grįžtamojo kvietimo modelio klasių diagrama.

Klasių aprašymai:

1. *ProactorServer* – pagrindinė modelio klasė atsakinga už ryšio užmezgėjo objekto sukūrimą bei to objekto darbo paleidimą.

2. *Acceptor* – ryšio užmezgėjas. Klasė atsakinga už ryšio užmezgimą, užklausų apdorotojų sukūrimą bei už jų grįžtamųjų metodų užregistravimą asinchroninių operacijų vykdymo kontroleriuose..
3. *RequestHandler* – užklausų apdorotojas. Klasė atsakinga už užklausos apdorojimą.
4. *RequestHandlerBase* – bendra visoms konkurentiškumo valdymo modeliams užklausos vykdymo klasė, iš kurios yra paveldėtas užklausų apdorotojas.
5. *ICompletionListener* – operacijos užbaigimo klausytojas. Interfeisas turintis vienintelį abstraktų grįžtamojo kvietimo metodą, kurį realizuoja užklausos apdorotojas.
6. *IAsyncFuture* – asinchroninės operacijos vykdymo kontroleris. Klasė atsakinga už asinchroninės operacijos vykdymą.

Modelio veikimas yra parodytas sekos diagramoje pav. 38.



pav. 38 Vienos gijos ir grįžtamojo kvietimo modelio scenarijaus diagrama.

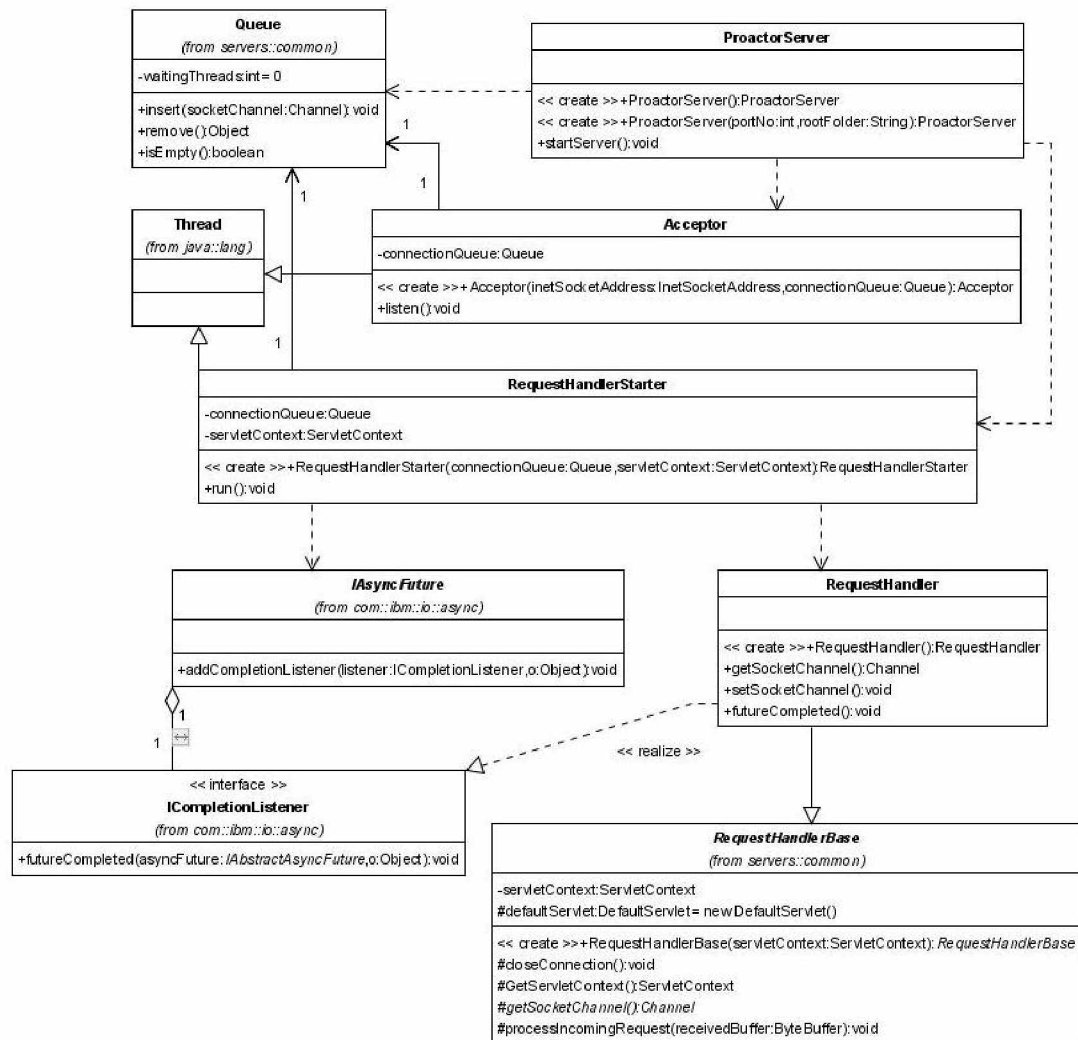
- Veiksmas prasideda nuo serverio paleidimo. Pirmajame žingsnyje sukuriamas yra *ProactorServer* objektas, kuris valdo pagrindinę programos eigą. Kuriant objektą parametrais yra perduodami porto numeris ir informacijos katalogas.
- Antrame žingsnyje yra sukuriamas ryšio užmezgėjo objektas (*Acceptor*). Kuriant objektą parametrais yra perduodami prisijungimo adreso bei informacijos konteksto objektai.
- Trečiame žingsnyje kviečiamas yra ryšio užmezgėjo metodas *listen*, kuris sukūrus savarankišką giją startuoja amžiną ciklą. Kiekvienos iteracijos metu yra atliekami tokie veiksmai:
  - Pradedama laukti naujų ryšių užmezgimo užklausų. Laukimo stadijoje gija užsiblokuoja kol neatsiras naujų ryšio užmezgimo užklausų.

  - Sulaukus ryšio užmezgimo užklausą yra užmezgamas prisijungimas.
  - Toliau yra asinchroniškai kviečiamas metodas *read*, kuris grąžiną asinchroninės operacijos vykdymo kontrolerį.
  - Sukuriamas užklausos apdorotojo objektas (*RequestHandler*). Kuriant objektą parametrais yra perduodami prisijungimo bei informacijos konteksto objektai.
  - Asinchroninės operacijos vykdymo kontroleryje yra užregistruojamas grįžtamasis užklausos apdorotojo kvietimas.
- Kadangi skaitymo/rašymo operacijos yra vykdomos asinchroniškai tolesnius veiksmus inicijuoja operacinė sistema, t.y. užbaigus kiekvieną asinchroninę operaciją operacinė sistema kviečia grįžtamąjį metodą *futureCompleted*.
  - Minėtą metodą realizuoja užklausų apdorotojai ir tokiu būdu toliau vyksta užklausų apdorojimas.

2.3.2. Atskiros gijos ryšio užmezgėjui, gijos prisijungimo eilės apdorojimui ir asinchroniškų užklausos apdorotojų su grįžtamuoju kvietimu modelis.

Šio modelio skirtumą nuo aprašyto praeitame skyriuje modelio sudaro prisijungimo eilės buvimas. Laiko tarpas nuo prisijungimo sukūrimo iki užklausų apdorotojo užregistravimo asinchroninės operacijos vykdymo kontroleryje teoriškai gali susidaryti toks, per kurį susikaupus ribiniam prisijungimo užklausų kiekiui operacinės sistemos lygmenyje būtų numetinėjamos visos naujai ateinančios prisijungimo užklausos. Todėl šis modelis turėtų sumažinti numetinėjamų prisijungimų skaičių esant dideliam konkurentiškumą valdančio komponento apkrovimui.

Realizuojamo modelio struktūra yra nurodyta paveiksle pav. 39.



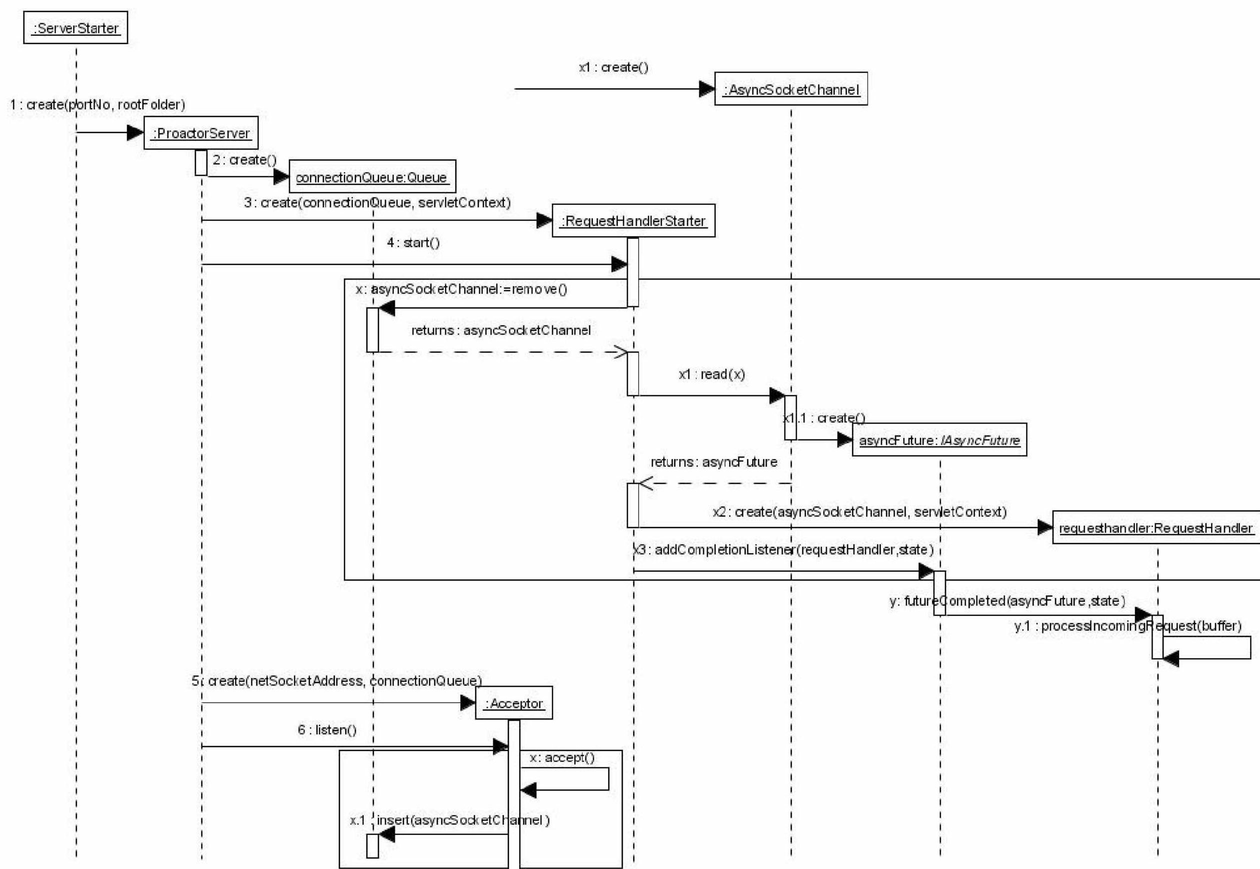
pav. 39 Atskiros gijos ryšio užmezgėjui, gijos prisijungimo eilės apdorojimui ir asinchroniškų užklausos apdorotojų su grįžtamuoju kvietimu modelio klasių diagrama.

Klasių aprašymai:

1. *ProactorServer* – pagrindinė modelio klasė atsakinga už ryšio užmezgėjo objekto ir užklausų apdorotojų kūrėjo sukūrimą bei atitinkamų objektų darbo paleidimą.
2. *Acceptor* – ryšio užmezgėjas. Klasė atsakinga už ryšio užmezgimą ir prisijungimų pridėjimą prie bendros prisijungimo eilės.
3. *Queue* – bendra prisijungimų eilė. Klasė atsakinga už prisijungimų kaupimą.
4. *RequestHandlerStarter* - užklausų apdorotojų kūrėjas. Klasė atsakinga už užklausų apdorotojų sukūrimą naudojant bendrą prisijungimo eilę bei už jų grįžtamųjų metodų užregistravimą asinchroninių operacijų vykdymo kontroleriuose.
5. *RequestHandler* – užklausų apdorotojas. Klasė atsakinga už užklausos apdorojimą.
6. *RequestHandlerBase* – bendra visoms konkurentiškumo valdymo modeliams užklausos vykdymo klasė, iš kurios yra paveldėtas užklausų apdorotojas.

7. *ICompletionListener* – operacijos užbaigimo klausytojas. Interfeisas turintis vienintelį abstraktų grįžtamojo kvietimo metodą, kurį realizuoja užklausos apdorotojas.
8. *IAsyncFuture* – asinchroninės operacijos vykdymo kontroleris. Klasė atsakinga už asinchroninės operacijos vykdymą.

Modelio veikimas yra parodytas sekos diagramoje pav. 40.



pav. 40 Atskiros gijos ryšio užmezgėjui, gijos prisijungimo eilės apdorojimui ir asinchroniškų užklausos apdorotojų su grįžtamuoju kvietimu modelio scenarijaus diagrama.

- Veiksmas prasideda nuo serverio paleidimo. Pirmajame žingsnyje sukuriamas yra *ProactorServer* objektas, kuris valdo pagrindinė programos eigą. Kuriant objektą parametrais yra perduodami porto numeris ir informacijos katalogas.
- Antrame žingsnyje yra sukuriama tuščia bendra prisijungimų eilė.
- Trečiame žingsnyje yra sukuriamas užklausų apdorotojų kūrėjo objektas (*RequestHandlerStarter*). Kuriant objektą parametrais yra perduodami bendros prisijungimo eilės ir informacijos konteksto objektai.
- Ketvirtame žingsnyje kviečiamas yra užklausų apdorotojų kūrėjo metodas *start*, kuris sukūrus savarankišką giją startuoja amžiną ciklą. Kiekvienos iteracijos metu yra atliekami tokie veiksmai:
  - o Pradedama laukti naujų prisijungimų bendroje prisijungimų eilėje. Laukimo stadijoje gija užsiblokuoja kol prisijungimų eilė nebus papildyta.

  - Sulaukus eilėje naujo prisijungimo toliau yra asinchroniškai kviečiamas *read* metodas, kuris grąžiną asinchroninės operacijos vykdymo kontrolerį.
  - Sukuriamas užklausos apdorotojo objektas (*RequestHandler*). Kuriant objektą parametrais yra perduodami prisijungimo bei informacijos konteksto objektai.
  - Asinchroninės operacijos vykdymo kontroleryje yra užregistruojamas grįžtamasis užklausos apdorotojo kvietimas.
- Penktame žingsnyje yra sukuriamas ryšio užmezgėjo objektas (*Acceptor*). Kuriant objektą parametrais yra perduodami prisijungimo adreso bei bendros prisijungimo eilės objektai.
- Septintame žingsnyje kviečiamas yra ryšio užmezgėjo metodas *listen*, kuris sukūrus savarankišką giją startuoja amžiną ciklą. Kiekvienos iteracijos metu sinchroniškai yra atliekami tokie veiksmai:
  - Pradedama laukti naujų ryšių užmezgimo užklausų. Laukimo stadijoje gija užsiblokuoja kol neatsiras naujų ryšio užmezgimo užklausų.
  - Sulaukus ryšio užmezgimo užklausą užmegztas prisijungimas yra pridedamas prie bendros prisijungimų eilės.
- Kadangi skaitymo/rašymo operacijos yra vykdomos asinchroniškai tolesnius veiksmus inicijuoja operacinė sistema, t.y. užbaigus kiekvieną asinchroninę operaciją operacinė sistema kviečia grįžtamąjį metodą *futureCompleted*.
  - Minėtą metodą realizuoja užklausų apdorotojai ir tokiu būdu toliau vyksta užklausų apdorojimas.

2.3.3. Vienos gijos modelis su parametrizuota gijų grupe pasibaigusių asinchroninių operacijų rezultatų apdorojimui.

Šio modelio realizacija susideda iš visų pagrindinių komponentų, kurie įeina į D. Schmidt proaktoriaus projektavimo schemą. Šio modelio skirtumą nuo dviejų modelių aprašytų praeituose skyriuose sudaro darbinių gijų grupės bei pasibaigusių asinchroninių operacijų eilės buvimas. Nors teoriškai galima realizuoti minėtus skirtumus, bet tai tikrai neduos žymesnių pagreitėjimo rezultatų. Atvirkščiai turi pasijausti užklausų apdorojimo uždelsimas dėl vienos paprastos priežasties. Asinchroniškų operacijų vykdymas JAVA programavimo kalboje yra realizuojamas panaudojant šiuo metu vienintelę IBM sukurtą asinchroninių skaitymų/rašymų operacijų biblioteką. Šitos bibliotekos teikiamas interfeisas yra labiau orientuotas į grįžtamojo kvietimo naudojamą. Todėl kuriant gijų grupę ir pasibaigusių rezultatų eilę dirbtinai būtų sukurtas papildomas ir nereikalingas funkcionalumas, kurio realizacija vis tiek naudotų grįžtamojo kvietimo modelį. Dėl išvardintos priežasties šis modelis toliau nebus nagrinėjamas.

3. Konkurentiškumą valdančių modelių realizacijų tyrimas.

3.1. Tyrimo metodika

Pagrindinės metodikos pasirinktos tyrimo metu yra našumo testavimas ir apkrovos testavimas.

Aprašytos šiame darbe konkurentiškumą valdantys modeliai bus naudojami supaprastintame HTTP serveryje kaip komponentas atsakingas už ryšio užmezgimą, užklausos nuskaitymą bei rezultato išsiuntimą.

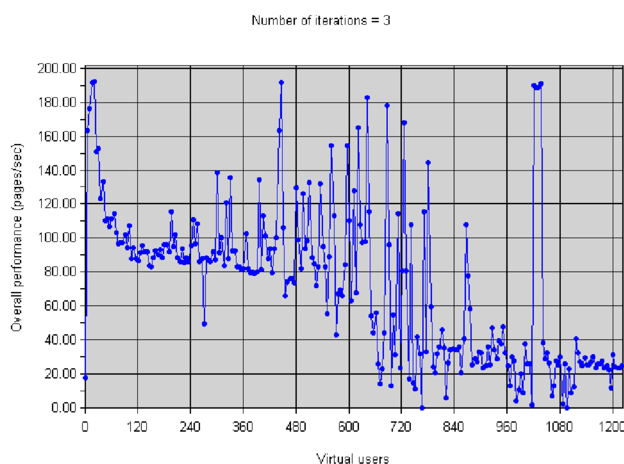
Našumo testavimas simuliuoja realų serverio naudojimą įprastomis sąlygomis, paprastai tam, kad įvertinti kaip serveris funkcionuos tam tikroje aplinkoje su fiksuotomis sąlygomis.

Apkrovimo testavimas yra skirtas aptikti kritinį apkrovimo lygį, su kuriuo serveris nepajėgia susidoroti. Testavimo metu yra generuojami dideli ir besikeičiančios apimties krovimai.

Vykdant testavimą bus nagrinėjami tokie matavimo vienetai:

- Bendras HTTP serverio našumas.

  Našumas yra apibrėžiamas kaip apdorotų užklausų kiekis per sekundę laiko, t.y. kiek serveris gali išsiusti prašomus puslapius vienos sekundes laikotarpyje. Šis matavimo vienetas sudaro viena iš grafiko koordinačių ašių, kur kita yra konkurentiškų naudotojų skaičius. Ašių susidūrimo taškuose bus matoma kiek užklausų yra apdorota esant tam tikram naudotojų skaičiui vienos sekundės metu. Grafiko pavyzdys yra pateiktas žemiau esančiame paveiksle pav. 41.

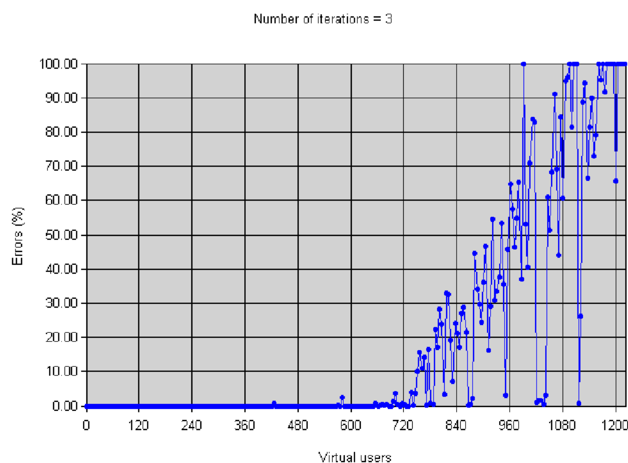


pav. 41 Bendro našumo grafiko pavyzdys.

- Procentinis klaidų kiekis.

  Procentinis klaidų kiekis parodo koks yra klaidų procentas nuo visų sėkmingai apdorotų užklausų. Šis matavimo vienetas yra vienas iš koordinačių ašių, kur kita ašis yra konkurentiškų naudotojų skaičius. Ašių susidūrimo taškuose bus matomas klaidų procentas nuo visų sėkmingai apdorotų užklausų siunčiamų tam tikro

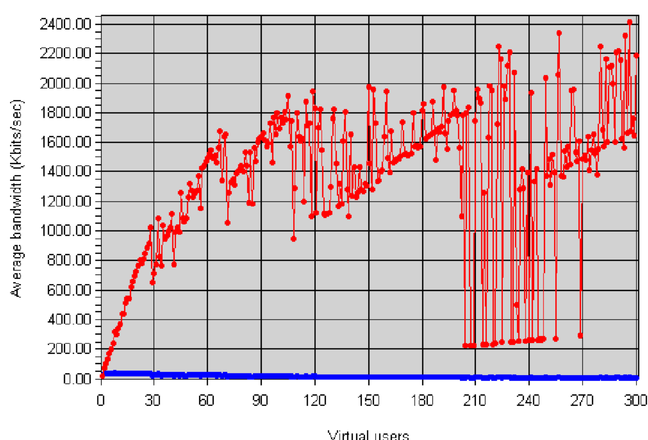
konkurentiškų naudotojų skaičiaus. Grafiko pavyzdys yra pateiktas žemiau esančiame paveiksle pav. 42.



pav. 42 Procentinio klaidų kiekio grafiko pavyzdys.

- Vidutinis HTTP serverio pralaidumas.

Pralaidumas yra apibrėžiamas kaip kliento vidutinis gautų bitų kiekis per sekundę laiko. Laiko matuoklis pradeda skaičiuoti nuo to laiko kai klientas pasiunčia HTTP užklausą. Tuo tarpu nustoja skaičiuoti uždarius prisijungimą kliento iniciatyvą. Gautų bitų skaičius susideda taip pat iš siunčiamų HTTP serverio antraštės informacijos. Šis matavimo vienetas yra vienas iš koordinačių ašių, kur kita ašis yra konkurentiškų naudotojų skaičius. Ašių susidūrimo taškuose bus matomas kiek vidutiniškai baitų per sekundę serveris sugeba išsiusti naudotojui esant tam tikram konkurentiškų naudotojų skaičiui. Grafiko pavyzdys yra pateiktas žemiau esančiame paveiksle pav. 43.

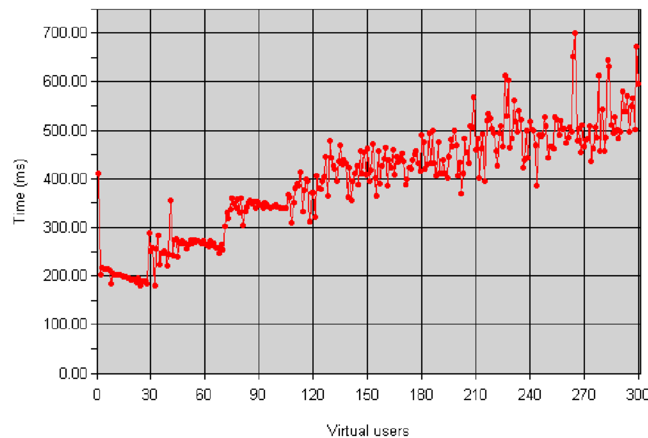


pav. 43 Vidutinio pralaidumo grafiko pavyzdys.

- Vidutinis HTTP klientų inicijuojamų transakcijų laikas, t.y. vidutinis gaišties laikas.

Gaišties laikas yra apibrėžiamas kaip vidutinis uždelsimų kiekis išreikštas milisekundėmis, kurį patiria klientas nuo užklausos išsiuntimo iki pilno atsakymo gavimo. Šitas matavimo vienetas parodo kaip ilgai galutinis Web naudotojas turi

laukti nuo HTTP GET užklausos išsiuntimo iki to laiko, kai jis pradeda gauti prašomą informaciją. Web transakcija tai laiko tarpas skaičiuojamas nuo pirmo užklausos baito išsiuntimo serveriui iki paskutinio gauto iš serverio atsakymo baito. Šis matavimo vienetas yra vienas iš koordinačių ašių, kur kita ašis yra konkurentiškų naudotojų skaičius. Ašių susidūrimo taškuose bus matomas kiek vidutiniškai reikia laukti naudotojui kol serveris pilnai aptarnaus jo prašoma užklausą esant tam tikram konkurentiškų naudotojų skaičiui. Grafiko pavyzdys yra pateiktas žemiau esančiame paveiksle pav. 44.



pav. 44 Vidutinio transakcijos laiko grafiko pazyzdys.

3.2. Tyrimo vykdymas

HTTP serveriui bus dedikuotas atskiras kompiuteris, kuriame bus paleistas tik jis vienas tam, kad eliminuoti ir minimizuoti galimų faktorių poveikį serverio darbui.

Kadangi tyrimo metu bus generuojami didelės apimties krovimai, dėl galimų techninių apribojimų sukuriant konkurentiškus vartotojus bus naudojamas daugiau negu vienas kompiuteris.

Našumo bei apkrovos testavime bus naudojama programa *WAPT* 3-ioji versija. *WAPT* tai krovimo, apkrovos ir našumo testavimo įrankis skirtas internetinių puslapių bei aplikacijų testavimui naudojantis žiniatinklio sąsaja (*web interface*) [SOF06].

Kiekviename kompiuteryje išskyrus tame, kur bus paleistas *HTTP* serveris, bus naudojami vienodi testavimo scenarijai, t.y. tam tikra veikla nukreipta į testuojamą serverį. Ją sudarys tam tikras kiekis virtualių vartotojų, kurie vykdys nustatytą transakcijų kiekį per nurodytą laiko tarpą.

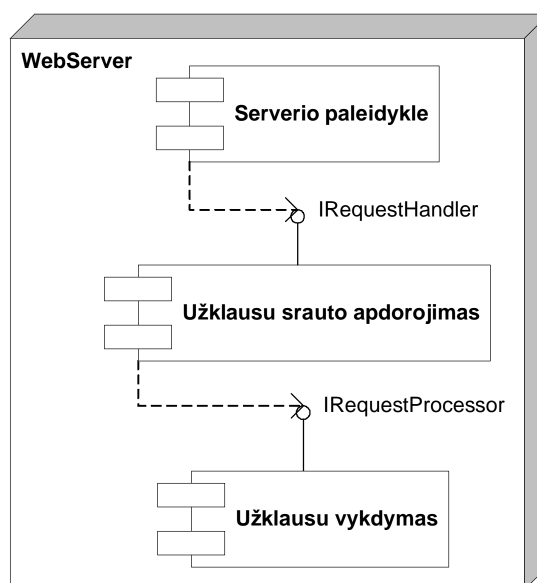
Tyrime bus naudojami du veiklos modeliai:

- **Pastovus krovimas –** fiksuotas virtualių vartotojų skaičius, kurie vykdo transakcijas per nustatytą laiko tarpą.
- **Augantis krovimas –** šitame modelyje testavimas prasideda nuo mažo virtualių vartotojų skaičiaus, kuris vėliau tam tikrais žingsniais auga iki nustatyto dydžio.

Kiekvienos iteracijos metu, virtualieji vartotojai įvykdo tam tikro resurso nustatytą kiekį kartų krovimą

3.3. Tyrimas

Tyrimo metu bus naudojamas sukurtas HTTP serveris, kurio komponentų diagrama yra pavaizduota paveiksle pav. 45. Testuojant kiekvieną iš aprašytų darbe modelių bus keičiamas tik užklausų srauto apdorojimo komponentas. Tokių būdu prasmingas tampa konkurentiškumo mechanizmų palyginimas, nes visas serverio darbas skirsis tik konkurentiškų užklausų apdorojimo realizacija.



pav. 45 Web serverio komponentų diagrama

Atliekamas tyrimas iš tikrųjų bus sudarytas iš trijų dalių:

- Pirmojoje dalyje bus testuojami baziniai konkurentiškumą valdantys modeliai. Šio tyrimo metu užklausų vykdymo komponentas nenaudos pagalbinės atminties objekto, kuriame galėtų būti patalpintos visos prašomos bylos. Taip yra daroma dėl to, kad užklausų srauto apdorojimo komponentas turėtų mažiau skaičiavimo išteklių, kurių dalys bus skirta užklausų vykdymo komponentui. Tokiu būdu užklausų srauto apdorojimo komponentas turės labiau įprastas ir būdingas HTTP serveriui veikimo sąlygas.
- Antroje dalyje bus testuojami bazinių konkurentiškumą valdančių modelių modifikacijos. Šio tyrimo metu bus naudojamas pagalbinės atminties objekto, kuriame bus talpinamos visos prašomos bylos. Taip yra daroma, nes šio tyrimo rezultatai turėtų parodyti kokią modelių modifikaciją geriausiai apdoroja užklausų srautą esant kaip įmanoma mažiausiam užklausų vykdymo darbui.

- Trečioje dalyje bus išrinkti iš antros tyrimo dalies geriausiai veikiantys modeliai ir dar karta bus praleisti testavimo scenarijai, bet pridedant papildomo “darbo” užklausų vykdymo komponentui. Su šio tyrimo pagalba bus bandoma pamatyti kaip veikia geriausiai užklausų srautą apdorojantys modeliai esant būdingoms HTTP serveriui sąlygomis. Minėtą papildomą darbą sudarys prašomos bylos krovimas iš disko nenaudojant sparčiosios atminties objekto, visų bylų esančių darbinėje direktorijoje surinkimas ir jų dydžio paskaičiavimas.

3.3.1. Tyrimo aplinka

Kompiuterio naudojamo *HTTP* serverio paleidimui konfigūracija:

- Procesorius: Intel Pentium 4 2.8GHz, su *Hyper Threading* technologija.
- Operatyvioji atmintis: 512Mb
- Tinklo adapteris: Intel Pro 100Mb
- Operacinė sistema: Windows 2000 SP4

Kompiuterių naudojamų krovimo testų paleidimui konfigūracija:

Kompiuteris #1:

- Procesorius: Intel Pentium 4 2.6GHz
- Operatyvioji atmintis:512Mb
- Tinklo adapteris: Intel Pro 100Mb
- Operacinė sistema: Windows XP SP2

Kompiuteris #2:

- Procesorius: Intel Celeron 1.7GHz
- Operatyvioji atmintis:512Mb
- Tinklo adapteris: Intel Pro 100Mb
- Operacinė sistema: Windows 2000 SP4

Kompiuteris #3:

- Procesorius: Intel Pentium 4 2.6
- Operatyvioji atmintis:512Mb
- Tinklo adapteris: Intel Pro 100Mb
- Operacinė sistema: Windows 2000 SP4

3.3.2. Tyrimo eiga

Pirmoje tyrimo dalyje bus vykdomi du testavimo scenarijai:

1. **Apkrovimo testavimas.**

Testavimas prasideda nuo vieno virtualaus naudotojo, kurių skaičius yra didinamas kiekvienos iteracijos metu sekančiais trimis naudotojais. Nustatyta naudotojų skaičiaus riba yra 1300. Kiekvienos iteracijos metu kiekvienas naudotojas tris kartus užkrauna 1 KB dydžio puslapį. Tarp kiekvienos testavimo iteracijos nėra nustatyto uždelsimo laiko. Tuo tarpu laikas tarp virtualių naudotojų darbo pradžios yra 1 milisekundė. Laikas skaičiuojamas yra nuo pirmo užklausos baito pasiuntimo iki paskutinio atsakymo į užklausą baito atsiuntimo. Taipogi yra nustatytas 120 sekundžių laukimo į atsakymą laikas.

2. **Našumo testavimas.**

Testavimas susideda iš fiksuoto 500 virtualių naudotojų skaičiaus, kurie krauna 1 KB dydžio puslapį per nustatytą 20 minučių laiko tarpą. Uždelsimas tarp virtualių naudotojų darbo pradžios yra 1 milisekundė. Laikas skaičiuojamas yra nuo pirmo užklausos baito pasiuntimo iki paskutinio atsakymo į užklausą baito atsiuntimo. Taipogi yra nustatytas 120 sekundžių laukimo į atsakymą laikas (*time out*).

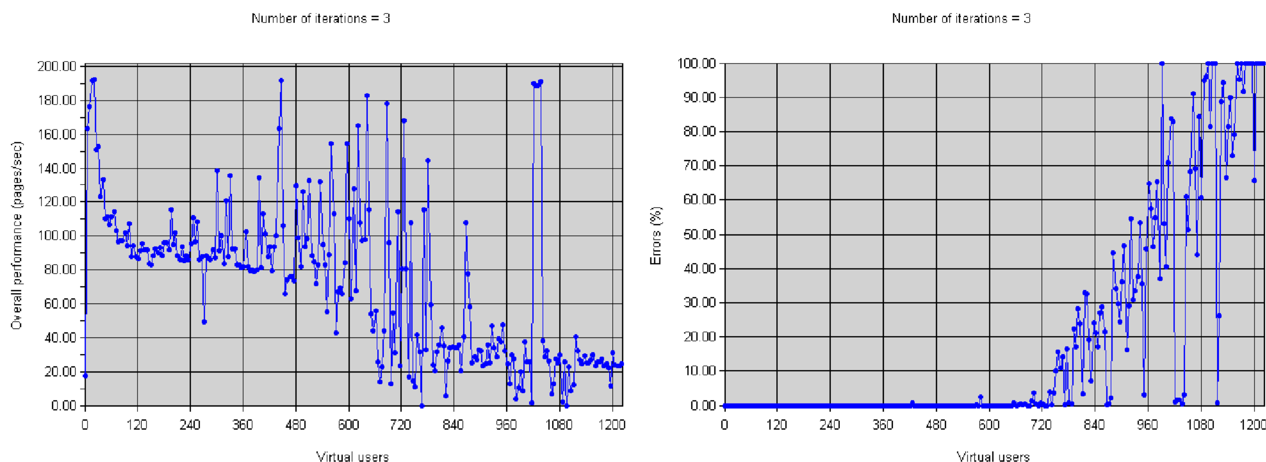
Antroje ir trečioje tyrimo dalyje serverio testavimas vyks su kintančiu virtualių naudotojų kiekiu, t.y. apkrovimas pradedamas nuo 1 naudotojo, pridedant po vieną didiname iki 300 konkurentiškai veikiančių naudotojų. Kiekvienam iš testuojamų modelių bus vykdomi atskiri testavimai kraudami kas kart skirtingo dydžio bylas, t.y. 500 baitų, 5 kilobaitų, 50 kilobaitų ir 500 kilobaitų. Tarp kiekvienos testavimo iteracijos nėra nustatyto uždelsimo laiko. Tuo tarpu laikas tarp virtualių naudotojų darbo pradžios yra 1 milisekundė. Taipogi yra nustatytas 120 sekundžių laukimo į atsakymą laikas.

Kadangi serveris bus apkraunamas dideliu kiekiu konkurentiškai veikiančiu naudotojų, dėl techninės įrangos galimybių apribojimo bei TCP/IP protokolo prisijungimo užbaigimo realizacijos yra pasirinkta minėtą 300 vienų metu veikiančių virtualių naudotojų ribą. Taip yra todėl, kad TCP/IP protokolas, pažymėjus prisijungimą kaip pasibaigusį, laiko pasyvų prisijungimą per 2MSL(angl. *Maximum Segment Life*) laiko tarpą, kuris *Windows* operacinėje sistemoje yra 240 sekundės. Dėl tokios priežasties sparčiai testuojant serverį yra tiesiog išnaudojami visi laisvi prisijungimo portai ir pradedamas prisijungimų užklausų numetinėjimas. Todėl prisijungimo užbaigimas buvo deleguotas Web klientui tokiu būdu užtikrinant, kad operacineje sistemoje, kurioje veikia Web serveris visada užtektų laisvu portų. Tuo tarpu Web kliento operacineje sistemoje buvo padindintas kuriamų portų numerių intervalas iki 65534.

3.4. Pirmos tyrimo dalies rezultatai.

Žemiau yra pateikiami vieno iš trijų naudojamų testavimo metu kompiuterių rezultatai. Todėl nagrinėjant diagramas reikia turėti omenyje, kad rezultatai yra sugeneruoti tris kart daugiau virtualių vartotojų negu yra nurodyta diagramose.

3.4.1. Gijos pagrindu realizuojamas valdomas konkurentiškumas.

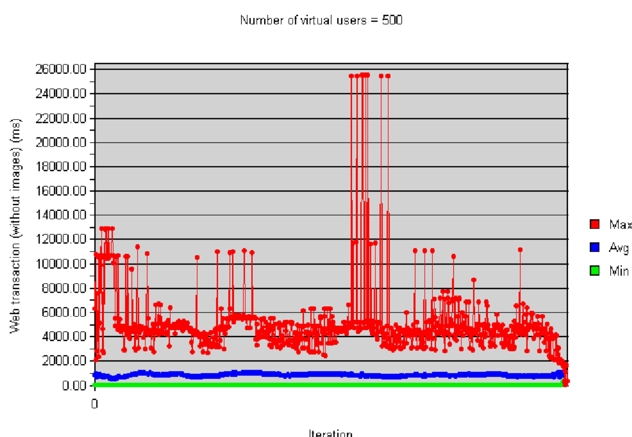


pav. 46 Gijos pagrindu realizuoto konkurentiškumo apkrovos testavimo našumo ir klaidų grafikai.

Procesoriaus apkrova testavimo metu svyravo ties 70%. Atminties sunaudojamas buvo apie 150Mb. Sukurtų maksimaliai vienu metu gijų kiekis buvo lygus 2750.

Kritiniai testavimo taškai:

- Iki 2000-2200 konkurentiškų virtualių vartotojų praktiškai nebuvo jokių klaidų
- Nuo 2000-2200 iki 3000 – klaidų kiekis tendencingai augo.
- Nuo 3000 *HTTP* serveris pradėjo metyti klaidas dėl atminties stokos.



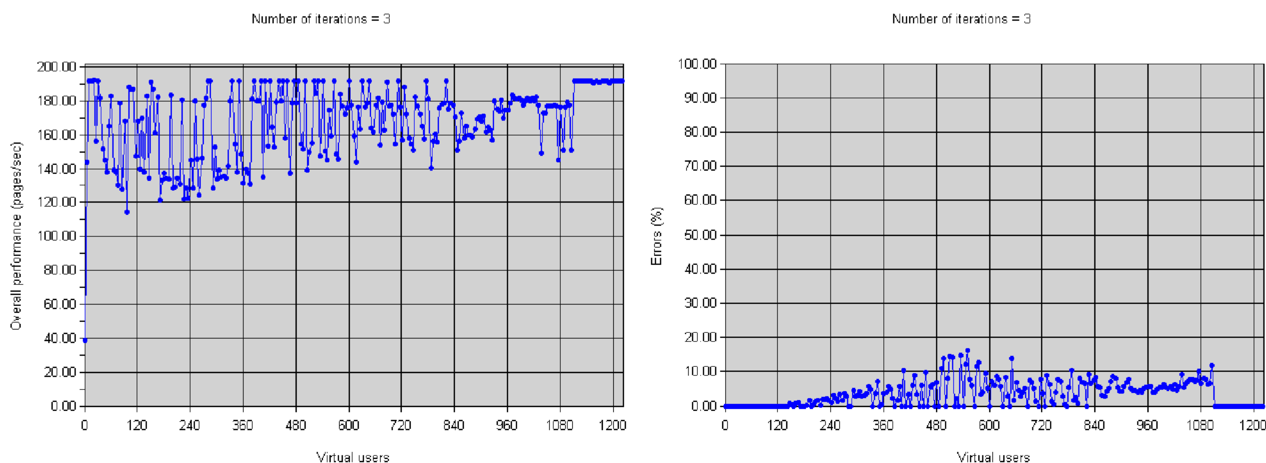
pav. 47 Gijos pagrindu realizuoto konkurentiškumo našumo testavimo rezultatai.

- Vidutinis web transakcijų greitis: 859,44 milisekundžių
- Įvykusių klaidų skaičius: 1435 (dėl susijungimo kanalų darbo)
- Vidutinis pakrautų puslapių per sekundę skaičius: 330,67
- Vidutinis vartotojų pakrautų puslapių skaičius: 794,196

Žiūrint į pateiktas diagramas, galima pastebėti našumo kritimo tendenciją didėjant konkurentiškų užklausų kiekiui. Tokią tendenciją galima paaiškinti tuo, kad kiekvienos užklausos apdorojimui yra naudojama atskira gija. Todėl didėjant virtualių vartotojų siunčiamų užklausų kiekiui, didėja ir gijų skaičius. Nors bendrai paėmus laikas reikalingas gijų konteksto perjungimui

nuo vieno iki kito žymiai neišauga, rezultate gautą našumo sumažėjimą sąlygojo tai, kad procesorius buvo tik vienas ir turėjo daug kartų persijunginėti tarp gijų ir tokiu būdu uždelsdamas kiekvienos gijos darbą.

3.4.2. Reaktoriaus schema

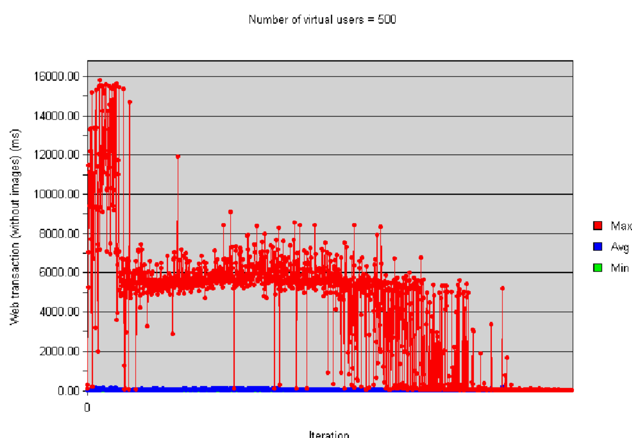


pav. 48 Reaktoriaus schemos realizacijos apkrovos testavimo našumo ir klaidų grafikai.

Procesoriaus apkrova testavimo metu svyravo ties 45%. Atminties sunaudojamas buvo apie 30Mb. Sukurtų maksimaliai vienu metu gijų kiekis buvo 1.

Kritiniai testavimo taškai:

- Iki 400-500 konkurentiškų virtualių vartotojų praktiškai nebuvo jokių klaidų
- Pradedant nuo 500 virtualių vartotojų klaidos pradėjo rodytis atsitiktinė tvarka, be jokios aiškios tendencijos, bet neviršijo 17%.

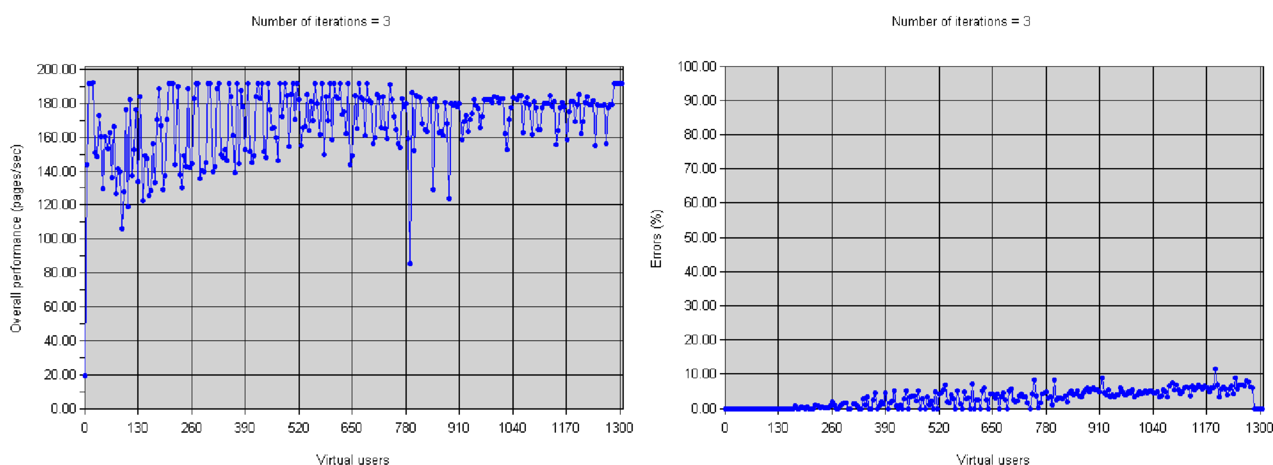


pav. 49 Reaktoriaus schemos realizacijos našumo testavimo rezultatai.

- Vidutinis web transakcijų greitis: 48,97 milisekundžių
- Įvykusių klaidų skaičius: 0
- Vidutinis pakrautų puslapių per sekundę skaičius: 421,55
- Vidutinis vartotojų pakrautų puslapių skaičius: 1016,312

Tyrimo metu nustatyta, kad vidutinis kraunamų puslapių skaičius sudaro apie 167,375 vienetų per sekundę, o įvykusių klaidų procentas vidutiškai svyruoja apie 6%. Beveik vienodą viso testavimo metu gaunamą krovimo našumą galima paaiškinti tuo, kad konkurentiškų užklausų apdorojimą vykdė tik vieną giją, kas eliminavo gijų kontekstų valdymą ankstesnio konkurentiškumo modelio atveju. Nors užklausų apdorojimas vyko naudojant nesiblokuojančią įvedimo/išvedimo posistemę, konkretūs užklausų apdorojimai vyko sinchroniškai, vienas po kito, kaip tarkim failo nuskaitymas iš disko. Šitas faktas gali būtį įvykusių klaidų priežastimi, nes apdorojant vieno vartotojo užklausą, o kitiems vartotojams bandant užmegzti ryšį gali atsitikti laukiančiųjų užmegzti ryšį eilės persipildymas ir kitų vartotojų užklausų atmetimas, kas yra laikoma klaidą. Kita vertus konkurentiškų užklausų apdorojimo greitis gali sumažėti dėl įvykių išskyrėjo darbo, nes didėjant prisijungimų skaičiui, didėja ir įvykių valdymas bei naujų įvykių apdorotojų registravimas selektoriuje.

3.4.3. Proaktoriaus schema

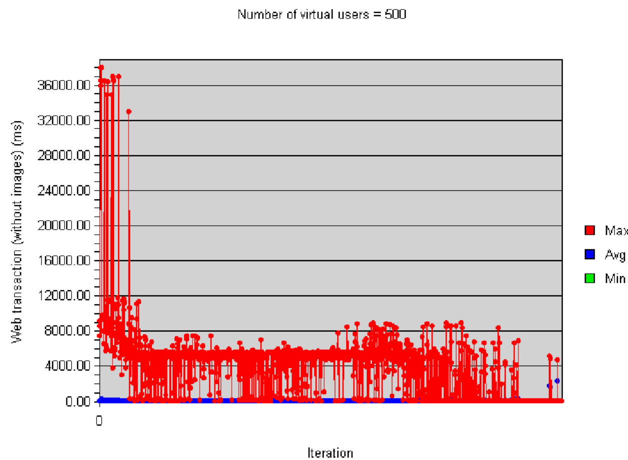


pav. 50 Proaktoriaus schemos realizacijos apkrovos testavimo našumo ir klaidų grafikai.

Procesoriaus apkrova testavimo metu svyravo ties 55%. Atminties sunaudojamas buvo apie 30Mb. Sukurtų maksimaliai vienu metu gijų kiekis buvo 1.

Kritiniai testavimo taškai:

- Iki 500 konkurentiškų virtualių vartotojų praktiškai nebuvo jokių klaidų
- Pradedant nuo 500 virtualių vartotojų klaidos pradėjo rodytis daugmaž stabiliai neviršijant 10% ribos.



pav. 51 Proaktoriaus schemos realizacijos našumo testavimo rezultatai

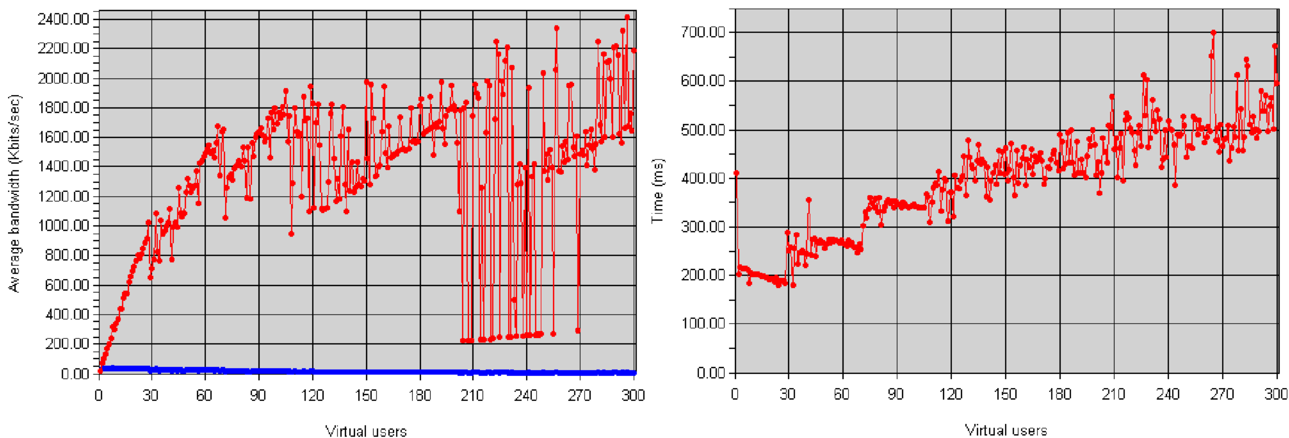
- Vidutinis web transakcijų greitis: 31,19 milisekundžių
- Įvykusių klaidų skaičius: 40 (dėl susijungimo kanalų darbo)
- Vidutinis pakrautų puslapių per sekundę skaičius: 426.81
- Vidutinis vartotojų pakrautų puslapių skaičius: 1027,544

Tyrimo metu nustatyta, kad vidutinis kraunamų puslapių skaičius sudaro apie 172,648 vienetų per sekundę, o įvykusių klaidų procentas vidutiškai svyruoja apie 4,5%. Palyginus su reaktoriaus, o tuo labiau su gijos pagrindu realizuotu konkurentiškumo rezultatais, matomas yra našumo pranašumas. Tokį rezultatą galėtų sąlygoti tai, kad, visu pirmą, konkurentiškumo komponentas darbą atlieką vienoje gijoje, tokiu būdu neužkraudamas procesoriaus nereikalingu gijų konteksto valdymu. Kita vertus nėra naudojamas įvykių išskyrėjas ir su juo susijęs įvykių valdymas. Visas pagrindinis darbas apdorojant konkurentiškų užklausų srautą yra praktiškai deleguojamas operacinei sistemai, o tai užtikrina greičiausias skaitymo ir rašymo operacijas.

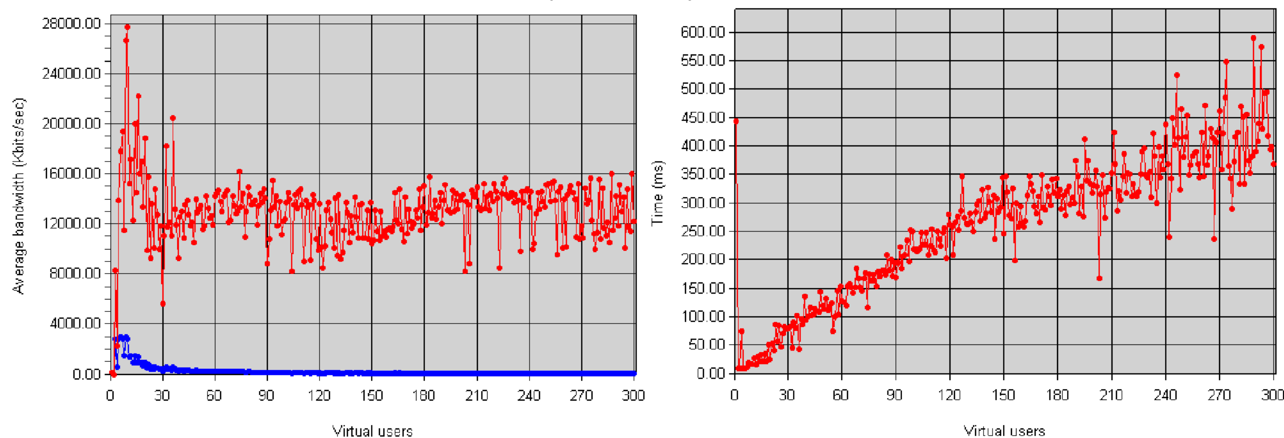
3.5. Antros tyrimo dalies rezultatai.

3.5.1. Sinchroninis besiblokuojantis užklausų apdorojimas.

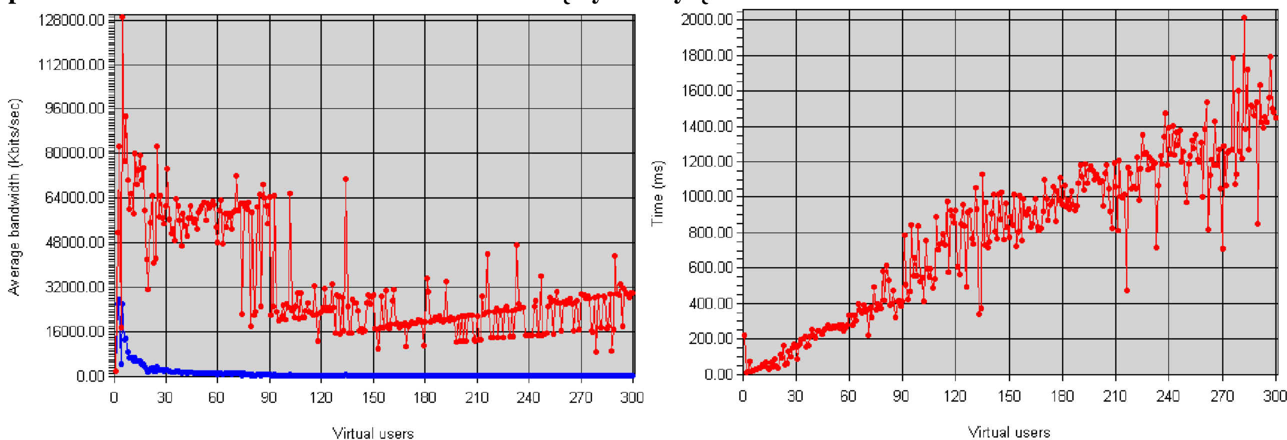
3.5.1.1. Vienos gijos vienam prisijungimui modelis.



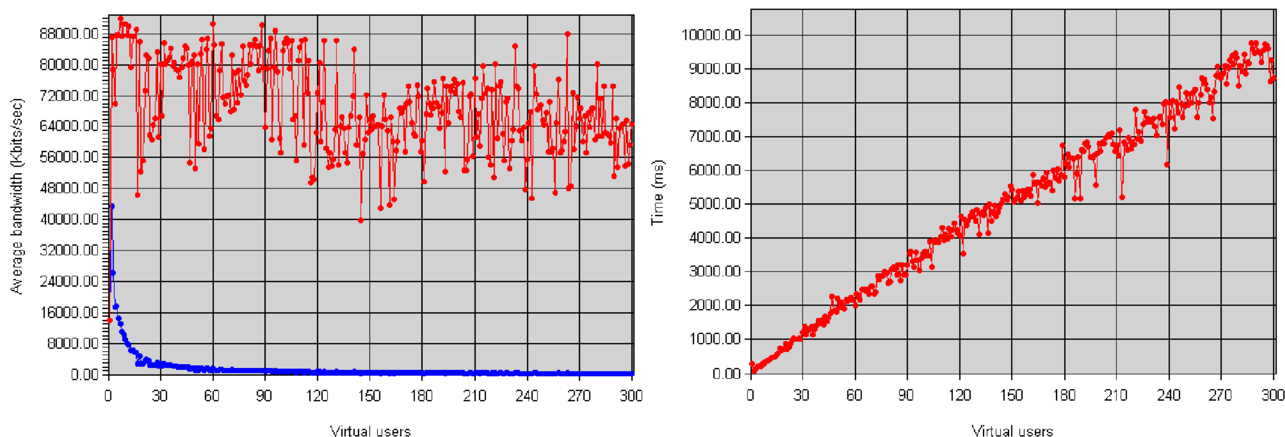
pav. 52 Testavimo rezultatai siunčiant 500 baitų dydžio bylą.



pav. 53 Testavimo rezultatai siunčiant 5 kilobaitų dydžio bylą.



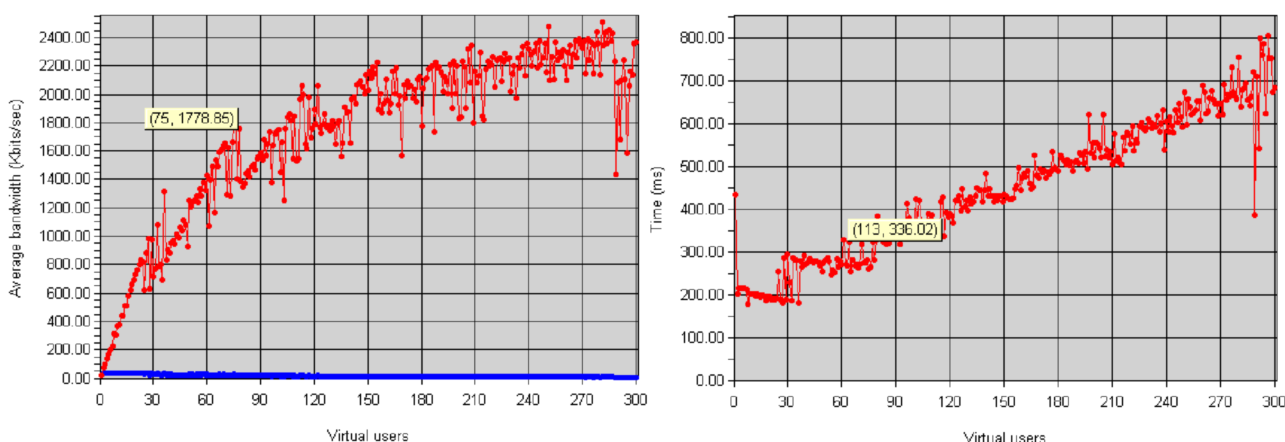
pav. 54 Testavimo rezultatai siunčiant 50 kilobaitų dydžio bylą.



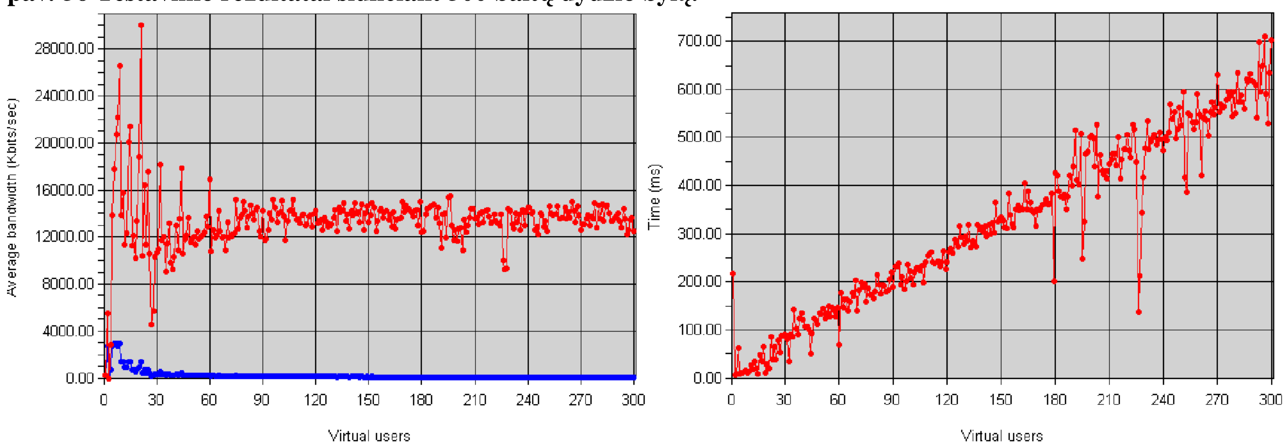
pav. 55 Testavimo rezultatai siunčiant 500 kilobaitų dydžio bylą.

Pateikti aukščiau rezultatai kaip ir buvo tikėtasi parodo, kad augant konkurentiškų klientų kiekiui vidutinis serverio pralaidumas krenta, o vidutinis transakcijos laikas auga. Pastebėtina dar tai, kad augant bylos dydžiui sparčiau išauga vidutinis transakcijos laikas.

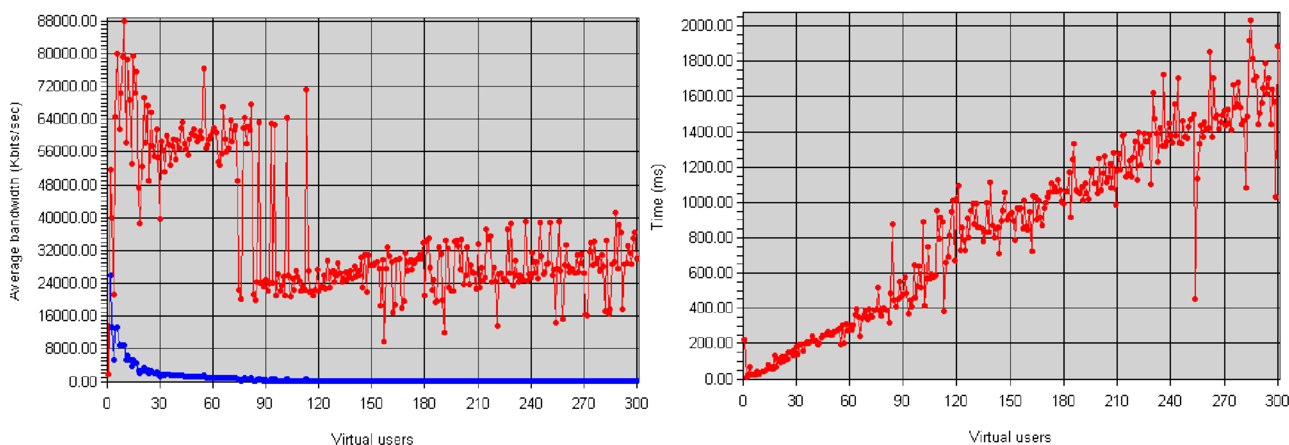
3.5.1.2. Vienos gijos vienam prisijungimui modelis su prisijungimo eile.



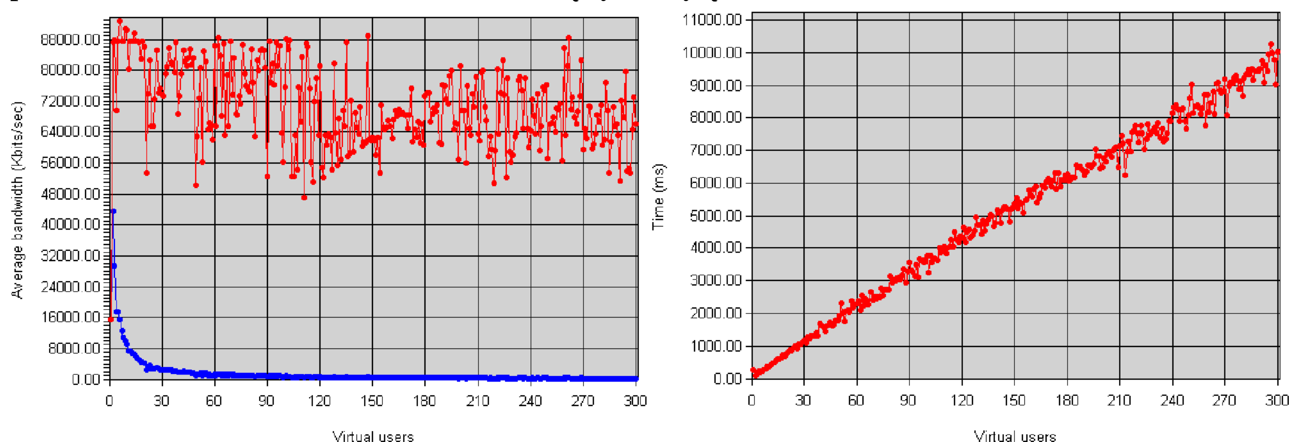
pav. 56 Testavimo rezultatai siunčiant 500 baitų dydžio bylą.



pav. 57 Testavimo rezultatai siunčiant 5 kilobaitų dydžio bylą.



pav. 58Testavimo rezultatai siunčiant 50 kilobaitų dydžio bylą.

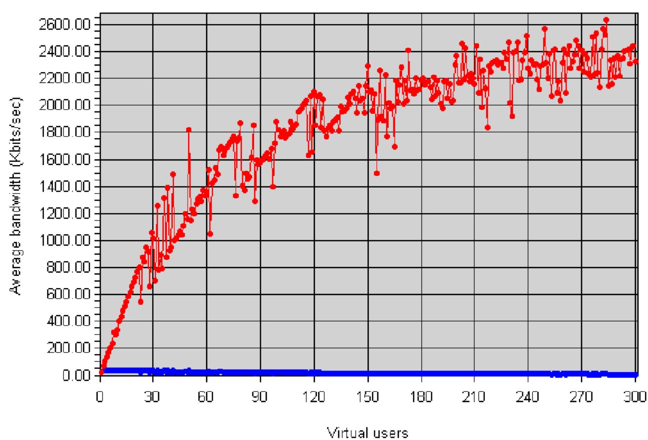


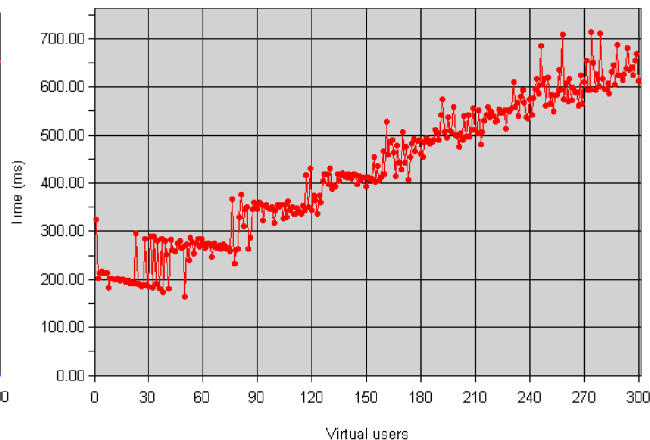
pav. 59 Testavimo rezultatai siunčiant 500 kilobaitų dydžio bylą.

Realizuojant šį konkurentiškumą valdantį modelį teoriškai buvo tikėtasi mažesnio ryšio užmezgimo užklausų atmetinėjimo negu vienos gijos vienam prisijungimui modelio atveju. Atlikus praktinį apkrovos testavimą buvo pastebėta, kad šio modelio naudai klaidų kiekis buvo mažesnis, nors klaidų kiekis abiejų modelių atveju buvo labai nedidelis. Ko gero padidinus virtualių naudotojų kiekiui klaidų kiekio skirtumas būtų didesnis.

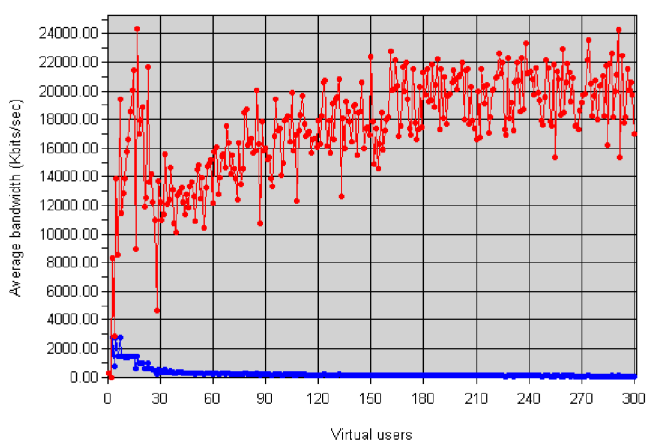
Nors testavimas parodė mažesnį klaidų skaičių, vidutinis transakcijos laikas šiek tiek išaugo lyginant su vienos gijos vienam prisijungimui modeliu. Tokio rezultato priežastimi galėtų būti papildomas procesoriaus laikas skirtas prisijungimų eilės apdorojimui.

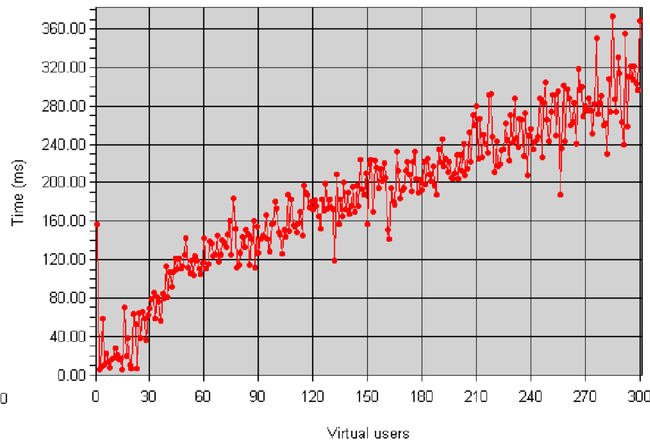
3.5.1.3. Vienos gijos vienam prisijungimui modelis su prisijungimo eile ir parametrizuota sukurtu gijų grupe.



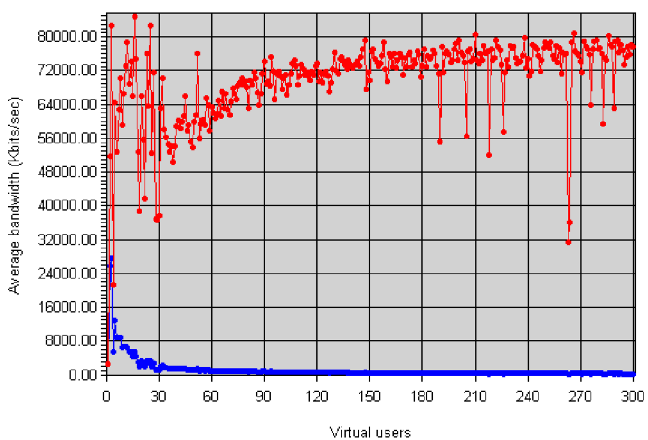


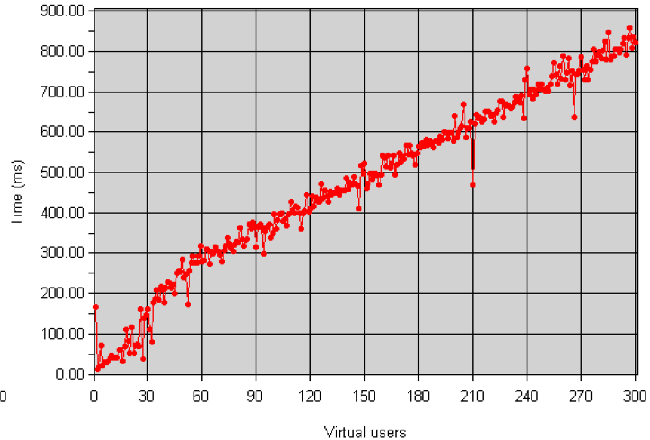
pav. 60 Testavimo rezultatai siunčiant 500 baitų dydžio bylą su 250 darbinėmis gijomis.



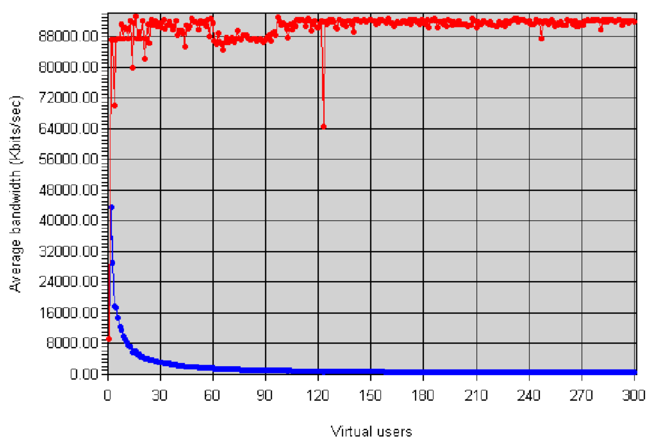


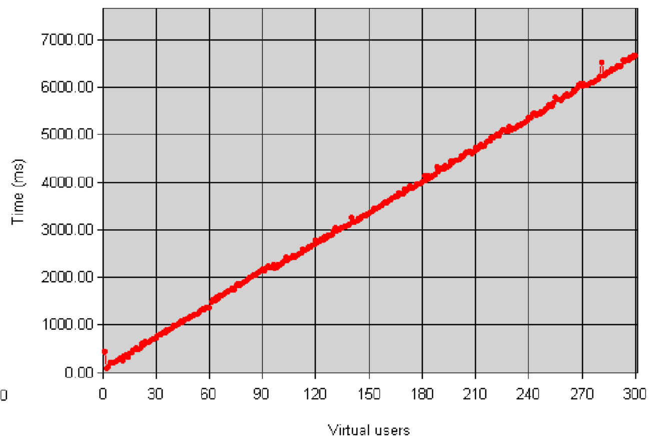
pav. 61 Testavimo rezultatai siunčiant 5 kilobaitų dydžio bylą su 50 darbinėmis gijomis.





pav. 62 Testavimo rezultatai siunčiant 50 kilobaitų dydžio bylą su 50 darbinėmis gijomis.





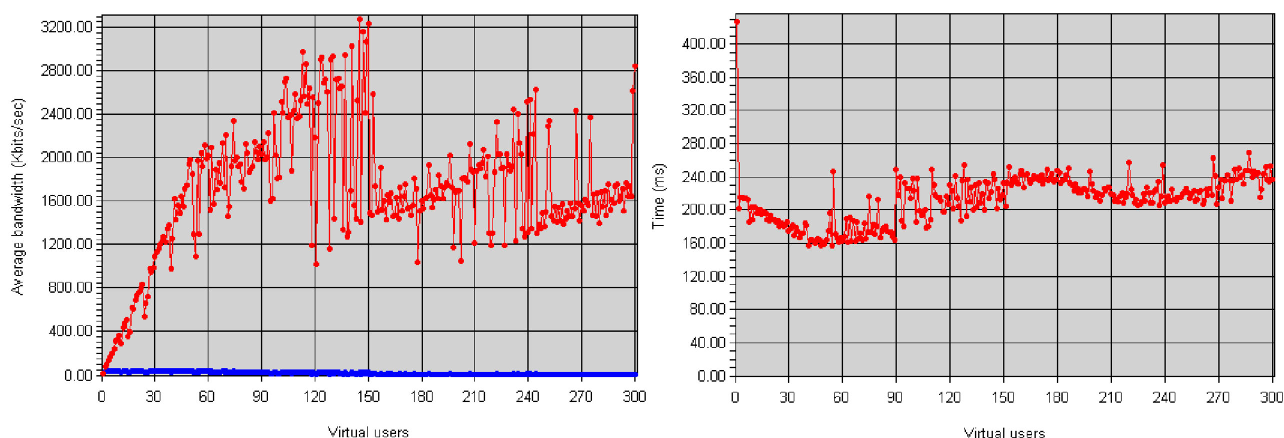
pav. 63 Testavimo rezultatai siunčiant 500 kilobaitų dydžio bylą su 5 darbinėmis gijomis.

Vykdant šio modelio apkrovimo testavimą, HTTP serveris buvo parametrizuojamas darbinių gijų skaičiumi. Atlikus testavimą pasirodo, kad siunčiant skirtingo dydžio bylas įtakos turi darbinių gijų skaičius, todėl pateikti aukščiau grafikai rodo geriausius rezultatus iš visų testavimo atveju. Pastebėtina, kad siunčiant mažesnio dydžio bylas turi būti didesnis darbinių gijų skaičius ir atvirkščiai siunčiant didesnio dydžio bylas reikalingas mažesnis darbinių gijų skaičius.

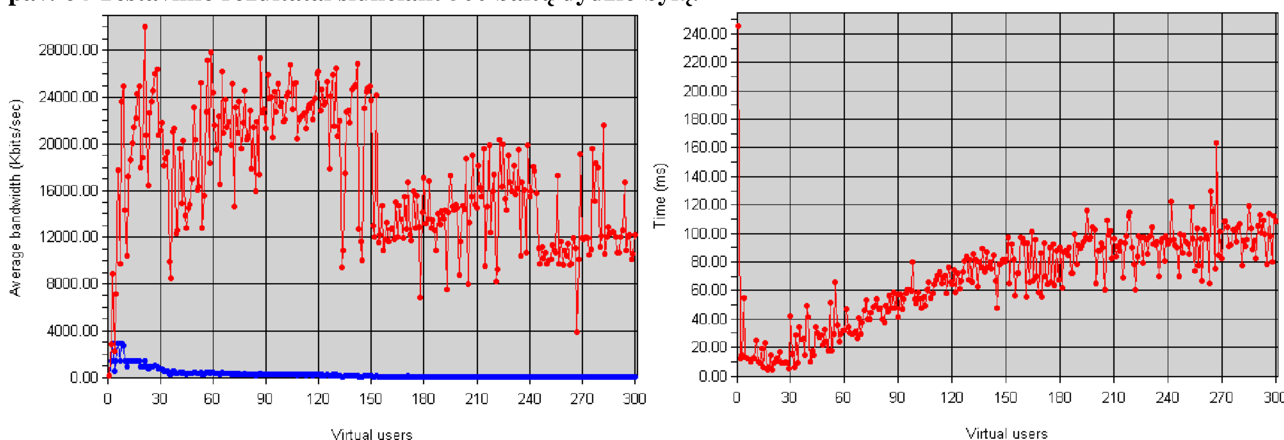
Realizuojant šį konkurentiškumą valdantį modelį teoriškai buvo tikėtasi sumažėjusio bendro užklausų apdorojimo laiko. Atlikus praktinį testavimą ir paliginus šio modelio su vienos gijos vienam prisijungimui modelio ir vienos gijos vienam prisijungimui modelio su prisijungimo eile rezultatais paaiškėjo, kad teorinė prielaida pasitvirtino. Dinaminis užklausų kūrimas sudaro žymią laiko dalį apdorojant užklausas. Mažas darbinių gijų kiekis palyginus su galimu daug didesniu vienų metu sukurtų dinamiškai gijų kiekių užklausas aptarnauja žymiai greičiau.

3.5.2. Sinchroninis nesiblokuojantis užklausų apdorojimas. Reaktoriaus schemos variantai.

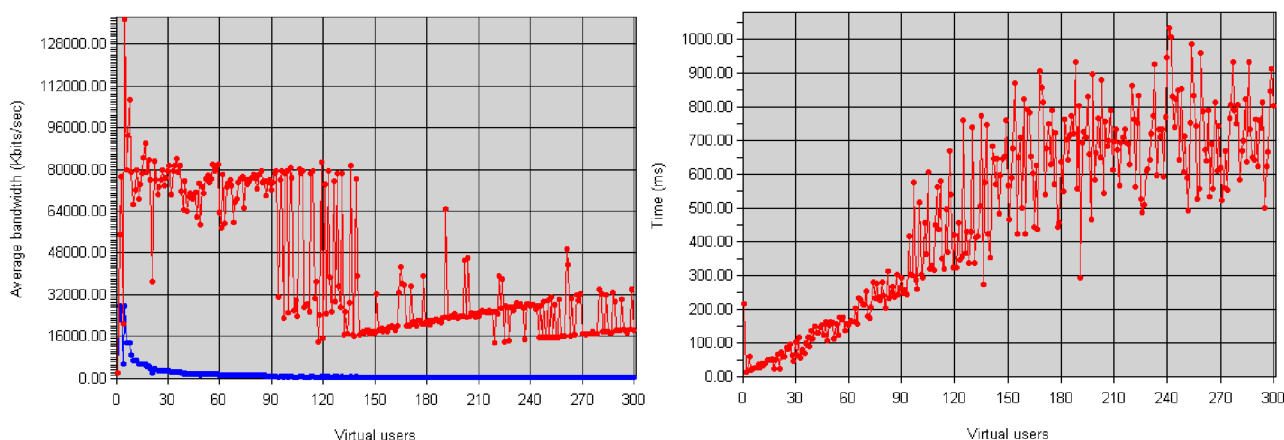
3.5.2.1. Vienos gijos ir vieno įvykių išskyrėjo modelis.



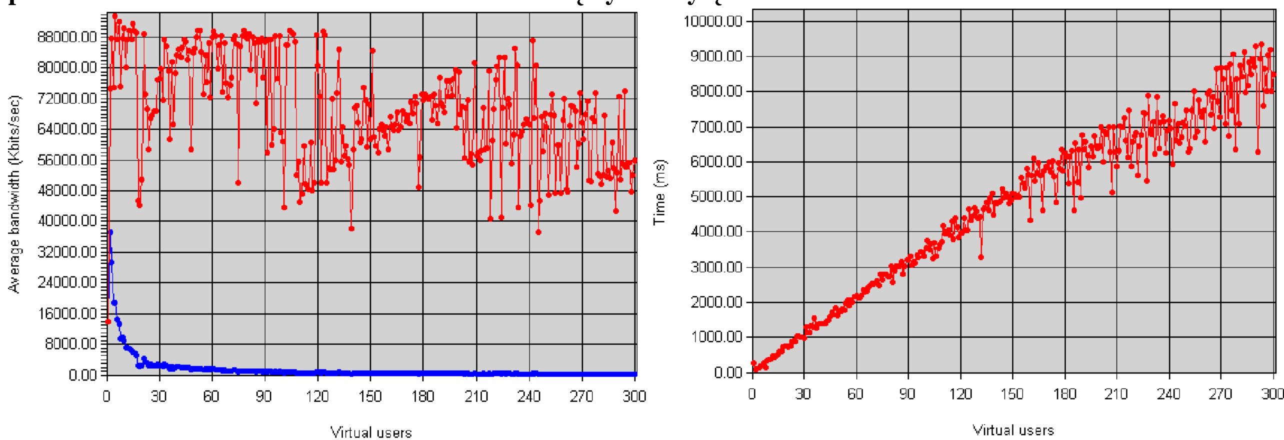
pav. 64 Testavimo rezultatai siunčiant 500 baitų dydžio bylą.



pav. 65 Testavimo rezultatai siunčiant 5 kilobaitų dydžio bylą.



pav. 66 Testavimo rezultatai siunčiant 50 kilobaitų dydžio bylą.



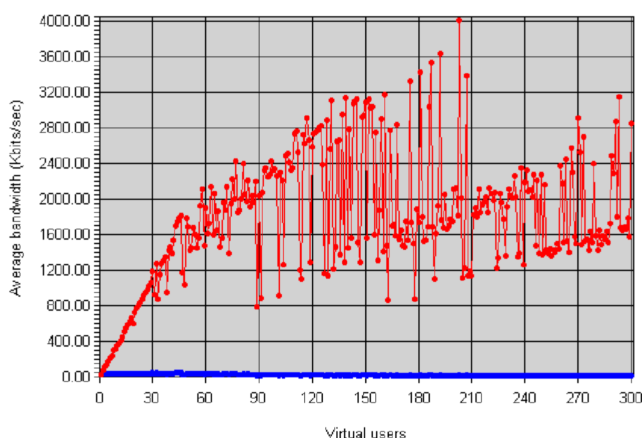
pav. 67 Testavimo rezultatai siunčiant 500 kilobaitų dydžio bylą.

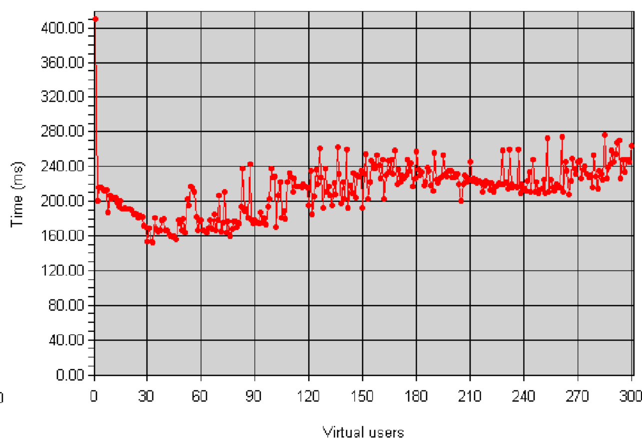
Pateikti rezultatai parodo tą pačią tendenciją kaip ir vienos gijos prisijungimui modelio atveju. Serverio pralaidumas augant konkurentiškai veikiančių klientų kiekiui mažėja, o vidutinis transakcijos laikas auga.

3.5.2.2. Atskiros gijos su įvykių išskyrėju ryšio užmezgėjui ir gijos užklausų apdorotojams modelis.

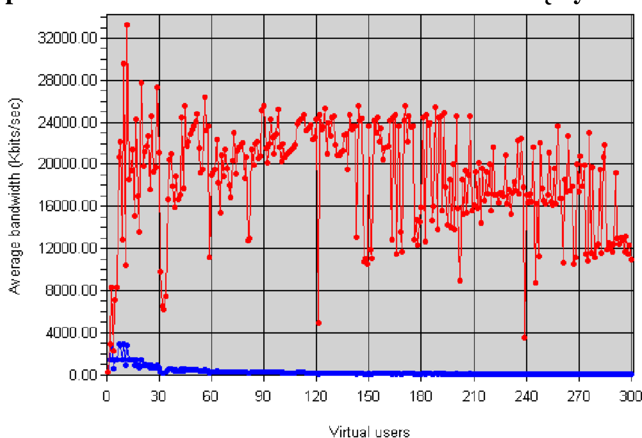
3.5.2.2.1. Modelio b atvejis.

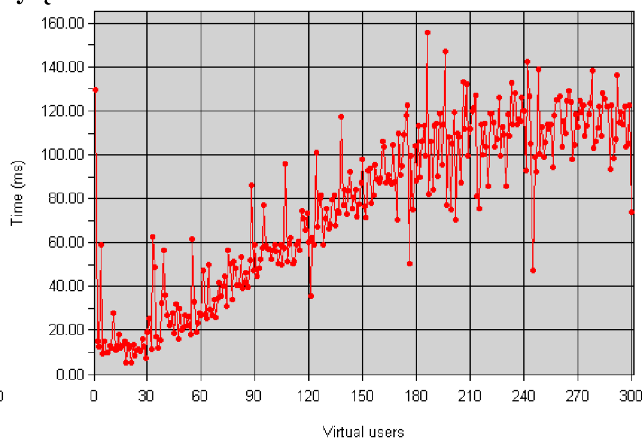
Abu įvykių išskyrėjai užsiblokuoja laukdami įvykių su vienodu laukimo nutraukimo laiku. Testavimo metu nagrinėjamas buvo toks laukimo laikas: 0 milisekundžių, t.y. laukimas nutraukiamas iškart, 1 milisekundė, 50 milisekundžių, 250 milisekundžių, 500 milisekundžių.



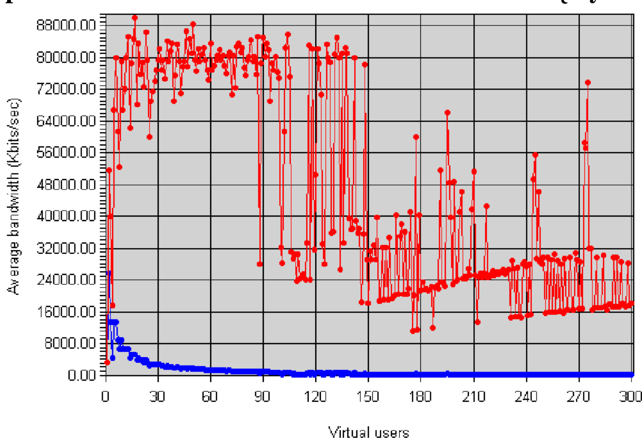


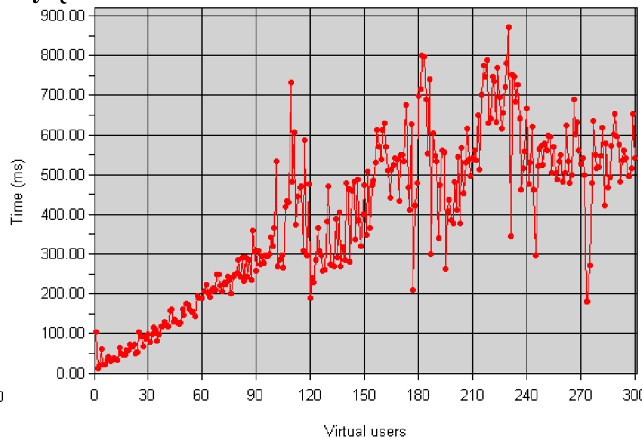
pav. 68 Testavimo rezultatai siunčiant 500 baitų dydžio bylą su 1 milisekundės laukimo laiku.



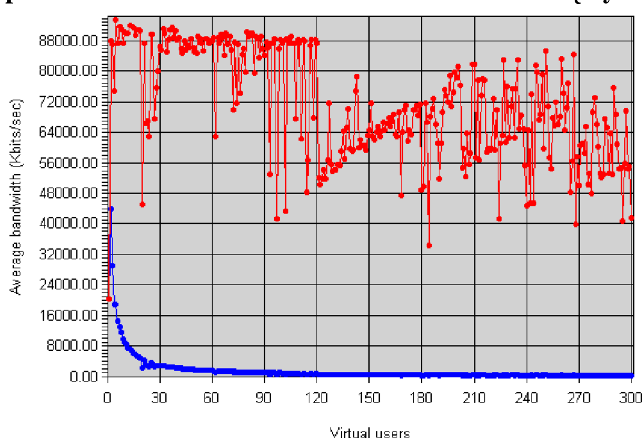


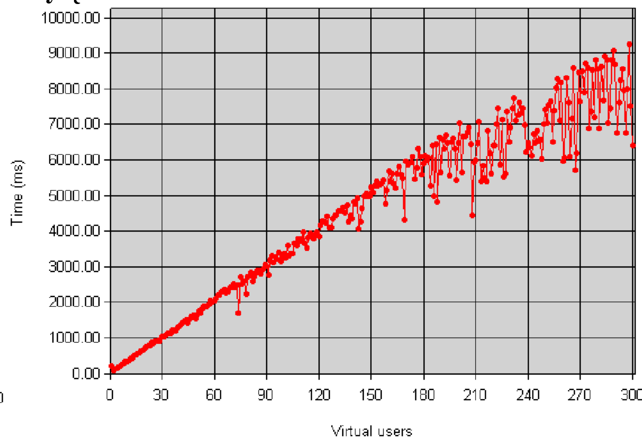
pav. 69 Testavimo rezultatai siunčiant 5 kilobaitų dydžio bylą su 1 milisekundės laukimo laiku.





pav. 70 Testavimo rezultatai siunčiant 50 kilobaitų dydžio bylą su 1 milisekundės laukimo laiku.

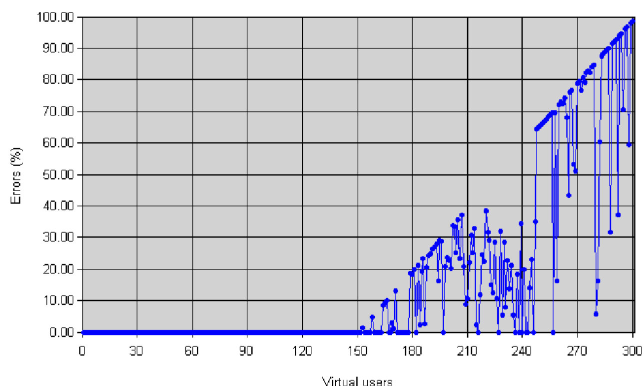




pav. 71 Testavimo rezultatai siunčiant 50 kilobaitų dydžio bylą su 0 milisekundžiu laukimo laiku.

Grafikai parodantys 50 kilobaitų dydžio failo siuntimo rezultatus yra rodomi testavimo su 1 milisekunde nepertraukiamo laukimo laiko, nors iš tikrųjų geresnį vidutinį transakcijos laiką parodė testavimas su 50 milisekundžių laukimo laiku. Kadangi pastarasis testavimas generuodavo daug daugiau ryšio užmezgimo klaidų, buvo nuspręsta, kad geresnį rezultatą duoda būtent testavimas su 1 milisekundės laukimu.

Vykdant šio modelio apkrovimo testavimą buvo pastebėta labai daug klaidų, kurios buvo ryšio užmezgimo numetinėjimo tipo. Didėjant nepertraukiamo laukimo laikui didėja klaidų kiekis ir vidutinis transakcijos laikas.

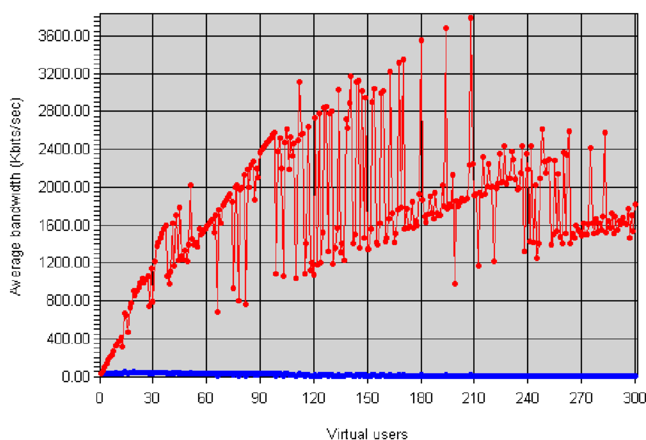


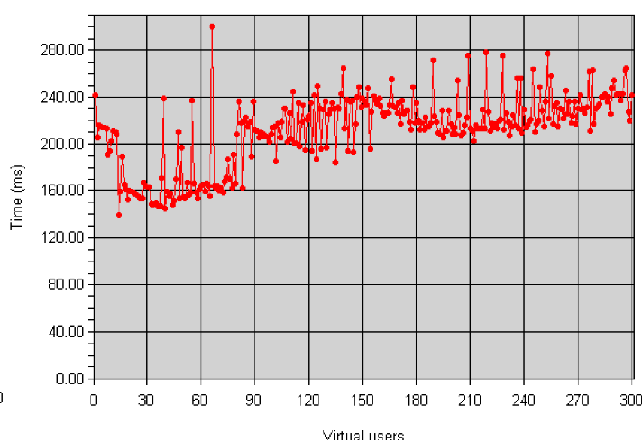
pav. 72 Generuojamos klaidos siunčiant 500 baitų dydžio byla esant 250 milisekundžių laukimo laikui.

Teorinė prielaida, kad sukūrus dvi atskiras gijas su savo įvykių išskyrėjais, vienas ryšio užmezgėjui, kitas užklausų apdorotojams sumažės numetinėjamų ryšio užmezgimo užklausų kiekis nepasitvirtino, o kaip parodė pateikti testavimo rezultatai klaidų atvirkščiai žymiai padidėjo. Lyginant vidutinį transakcijos laiką su vienos gijos ir vieno įvykių išskyrėjo modelio rezultatais matomas yra šio modelio sulėtėjimas.

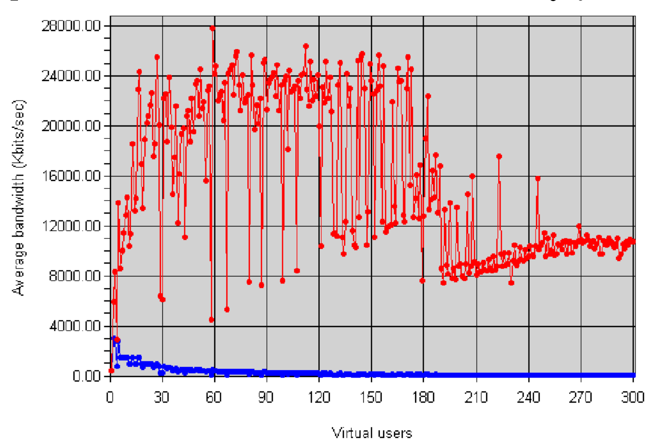
3.5.2.2.2. Modelio c atvejis.

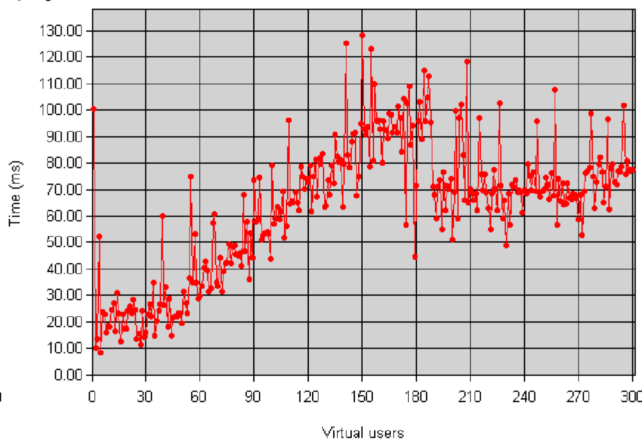
Vienas įvykių išskyrėjas užsiblokuoja tol kol atsiras prisijungimų įvykių. Tuo tarpu kitas įvykių išskyrėjas yra parametrizuojamas nepertraukiamo laukimo laiku. Testavimo metu nagrinėjamas buvo toks laukimo laikas: 0 milisekundžių, t.y. laukimas nutraukiamas iškart, 1 milisekundė, 50 milisekundžių, 250 milisekundžių, 500 milisekundžių.



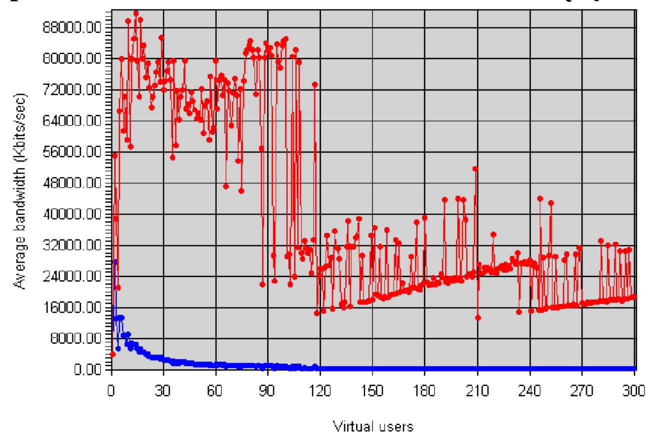


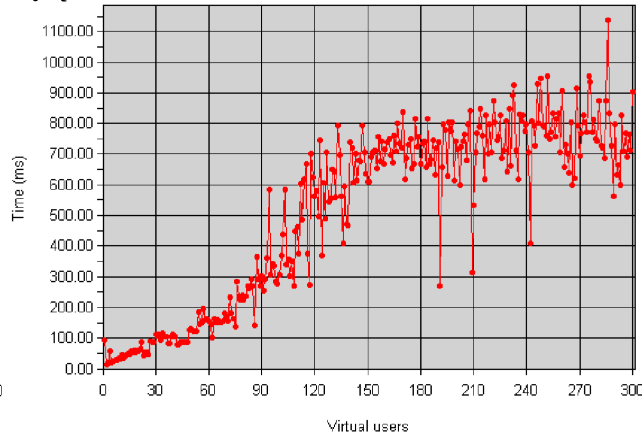
pav. 73 Testavimo rezultatai siunčiant 500 baitų dydžio bylą su 1 milisekundės laukimo laiku.



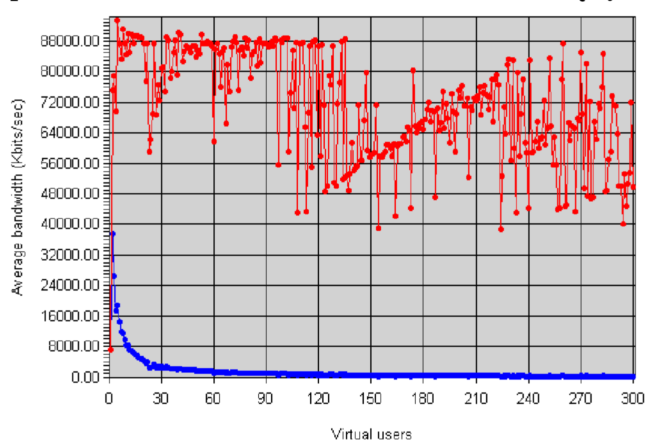


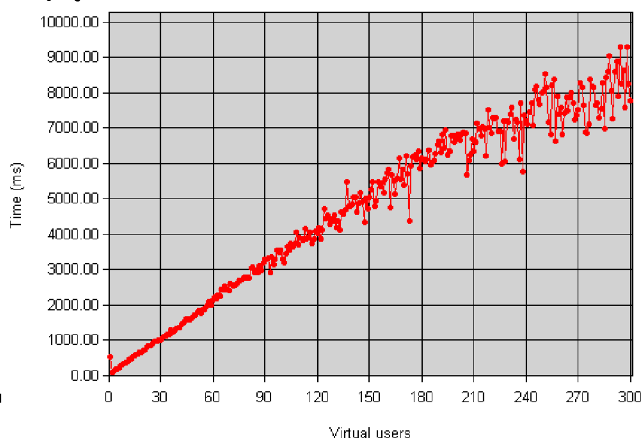
pav. 74 Testavimo rezultatai siunčiant 5 kilobaitų dydžio bylą su 1 milisekundės laukimo laiku.





pav. 75 Testavimo rezultatai siunčiant 50 kilobaitų dydžio bylą su 1 milisekundės laukimo laiku.

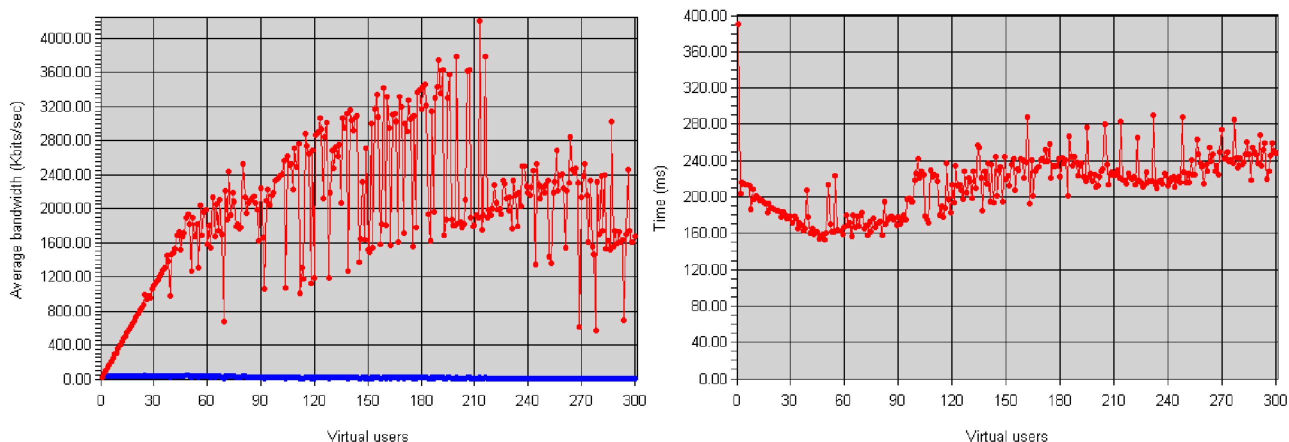




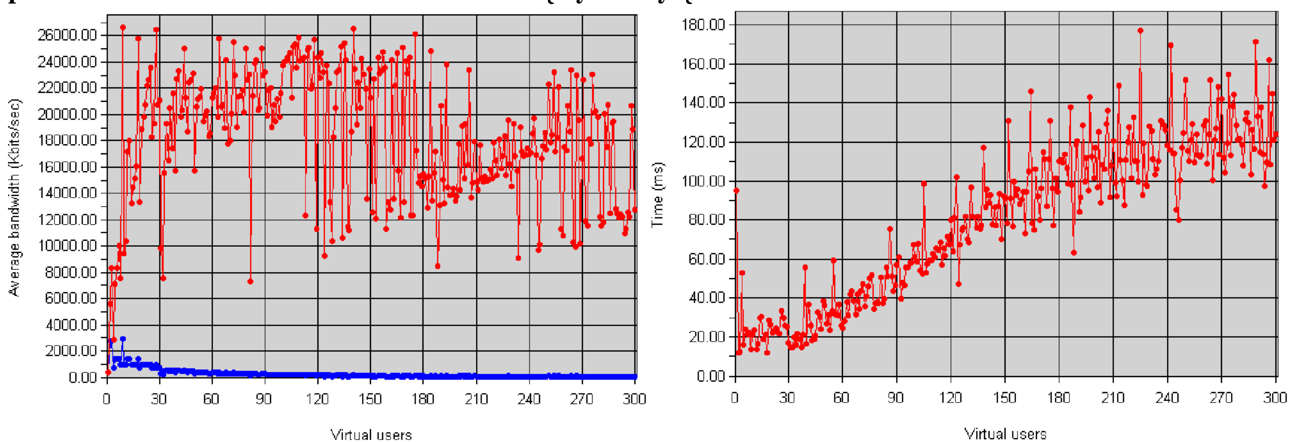
pav. 76 Testavimo rezultatai siunčiant 500 kilobaitų dydžio bylą su 1 milisekundės laukimo laiku.

Pagal aukščiau pateiktus rezultatus žymesnių skirtumų palyginus su šio modelio b atveju nesimato. Panašus yra ir vidutinis serverio pralaidumas, ir vidutinis transakcijos laikas. Nors klaidų atsiradimas šiek tiek sumažėjo, bet jų generavimas vykta pagal tą pačią taisyklę, kuo ilgesnis laukimo laikas ir konkurentiškai veikiančių naudotojų kiekis tuo daugiau klaidų.

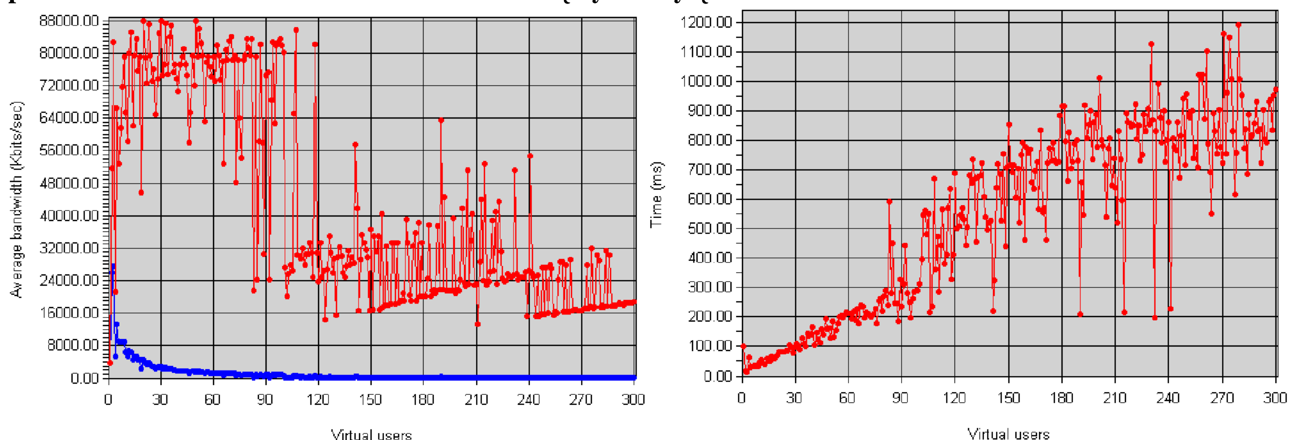
3.5.2.3. Atskiros gijos ryšio užmezgėjui ir gijos su įvykio išskyrėju užklausų apdorotojams modelis.



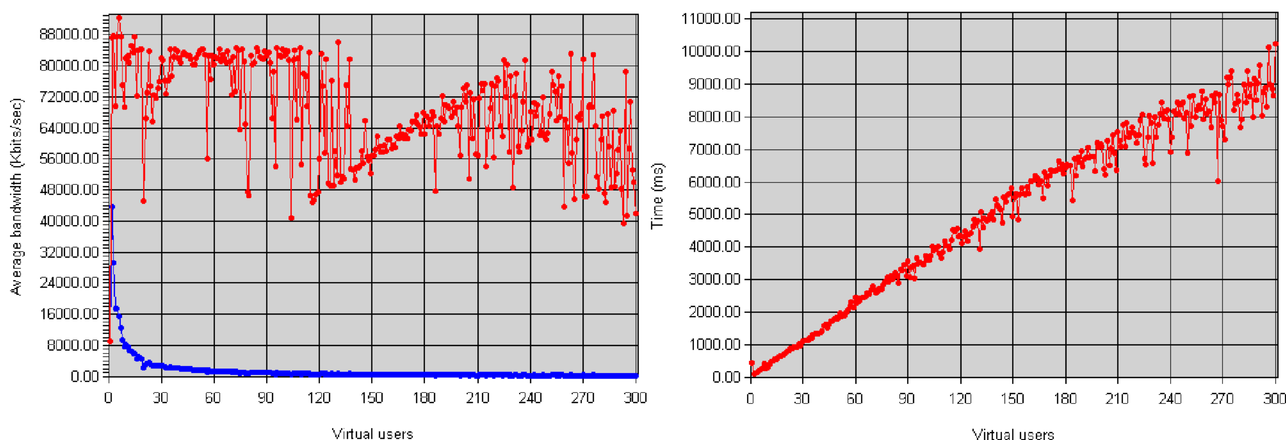
pav. 77 Testavimo rezultatai siunčiant 500 baitų dydžio bylą su 1 milisekundės laukimo laiku.



pav. 78 Testavimo rezultatai siunčiant 5 kilobaitų dydžio bylą su 1 milisekundės laukimo laiku.



pav. 79 Testavimo rezultatai siunčiant 50 kilobaitų dydžio bylą su 1 milisekundės laukimo laiku.

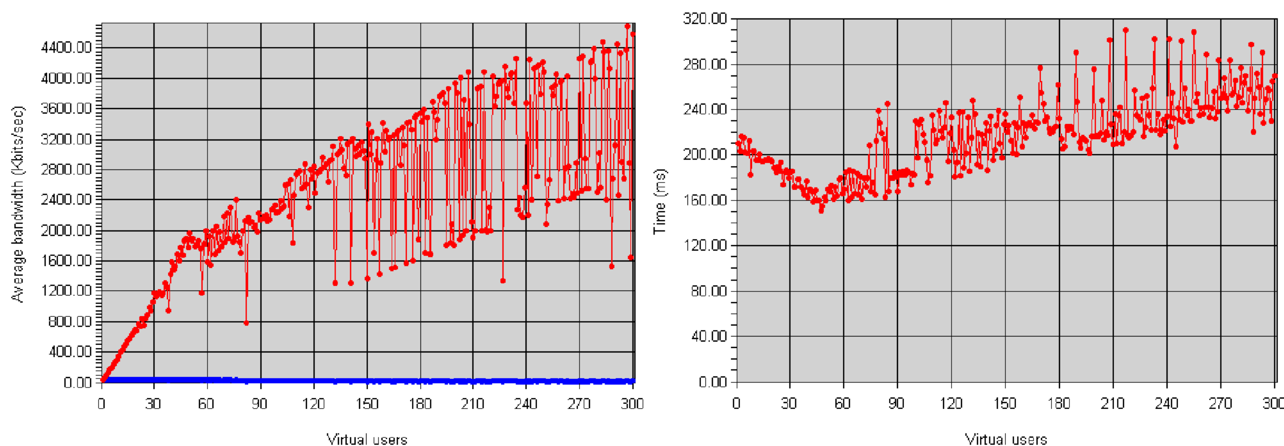


pav. 80 Testavimo rezultatai siunčiant 500 kilobaitų dydžio bylą su 1 milisekundės laukimo laiku.

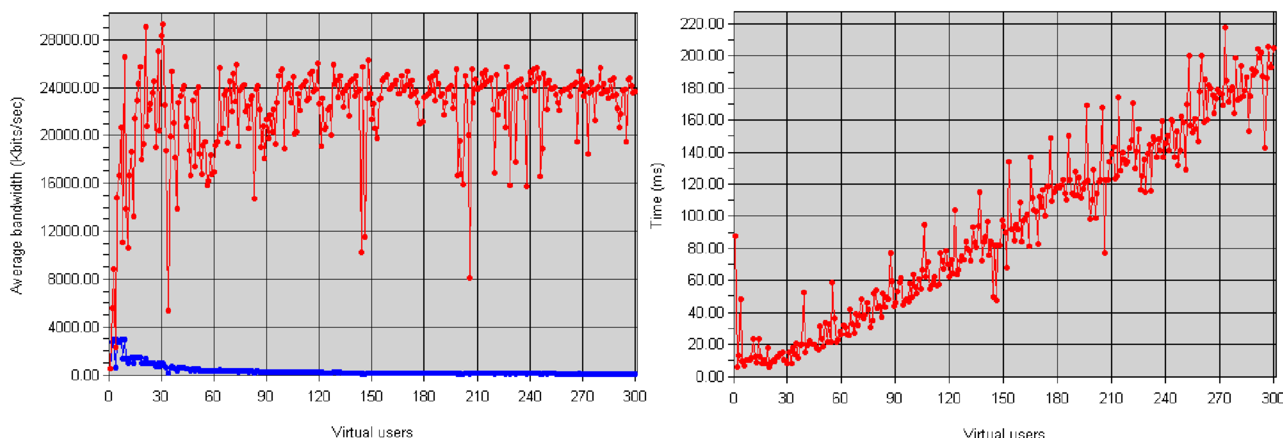
Teorinis šio modelio tikslas buvo įrodyti, kad naudojant atskirą giją su sinchroniniu besiblokuojančiu ryšio užmezgėju priešingai nuo atskiros gijos su nuosavų įvykių išskyrėju gali pagreitinti ryšio užmezgimą bei sumažinti atmetamų ryšio užmezgimo užklausų kiekį. Atlikus praktinį testavimą ir paliginus dviejų modelių gautus rezultatus žymesnio skirtumo nesimato. Vienas ir kitas modelis geriausiai veikė naudojant 1 milisekundės nepertraukiamo laukimo laiką. Matuojami kriterijai yra beveik panašūs.

Generuojamų klaidų kiekis rodo, kad šis modelis yra nestabilus kaip ir minėtas praeitame skyriuje. Tokį rezultatą galima paaiškini tuo, kad nors įvykių išskyrėjas pertraukia savo laukimą vienai milisekundei, bet to užtenka, kad prisijungimų eilė butų perpildytą ir tokiu būdų butų numetinėjamos ryšio užmezgimo užklausos.

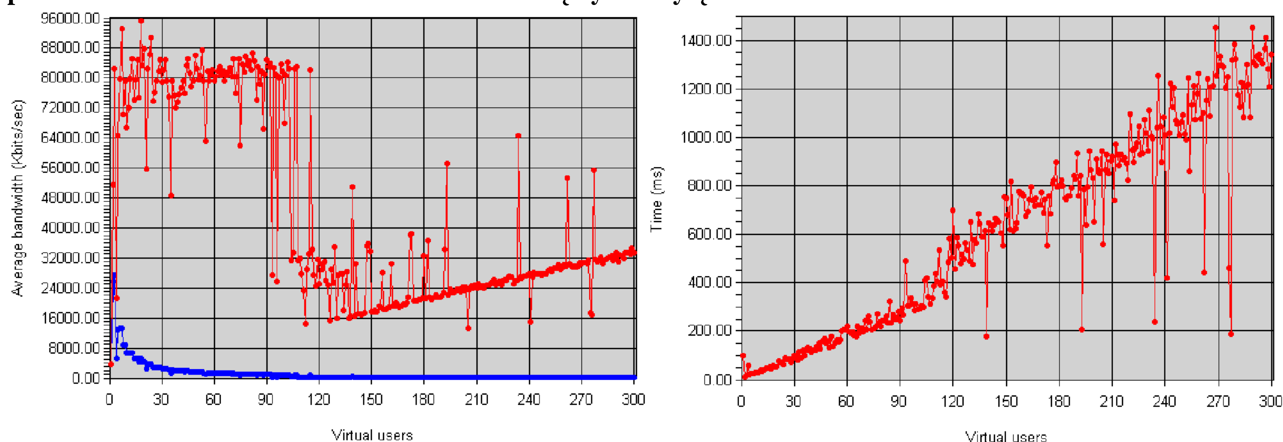
3.5.2.4. Atskiros gijos ryšio užmezgėjui, gijos prisijungimų eilės apdorojimui ir gijos su įvykio išskyrėju užklausų apdorotojams modelis.



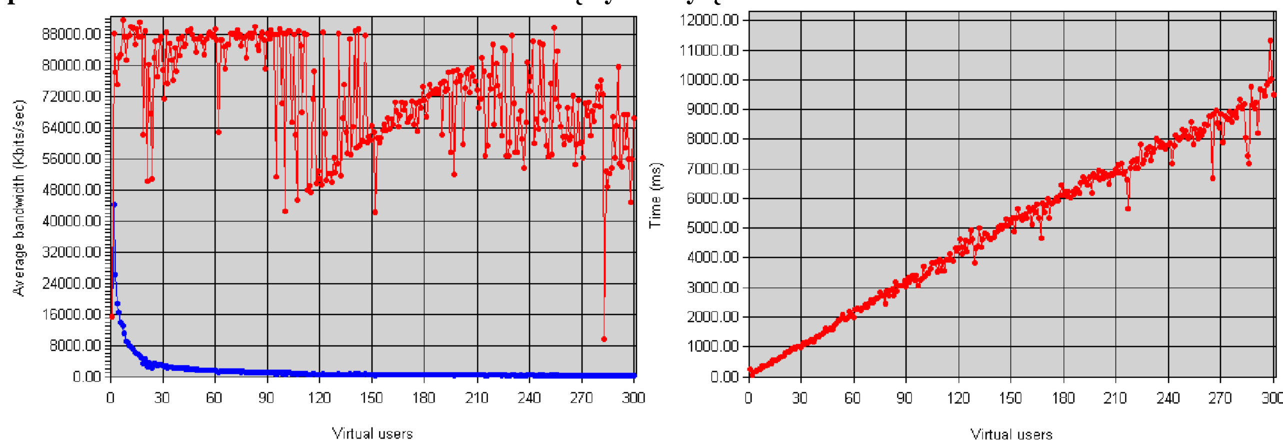
pav. 81 Testavimo rezultatai siunčiant 500 baitų dydžio bylą su 1 milisekundės laukimo laiku.



pav. 82 Testavimo rezultatai siunčiant 5 kilobaitų dydžio bylą su 1 milisekundės laukimo laiku.



pav. 83 Testavimo rezultatai siunčiant 50 kilobaitų dydžio bylą su 1 milisekundės laukimo laiku.

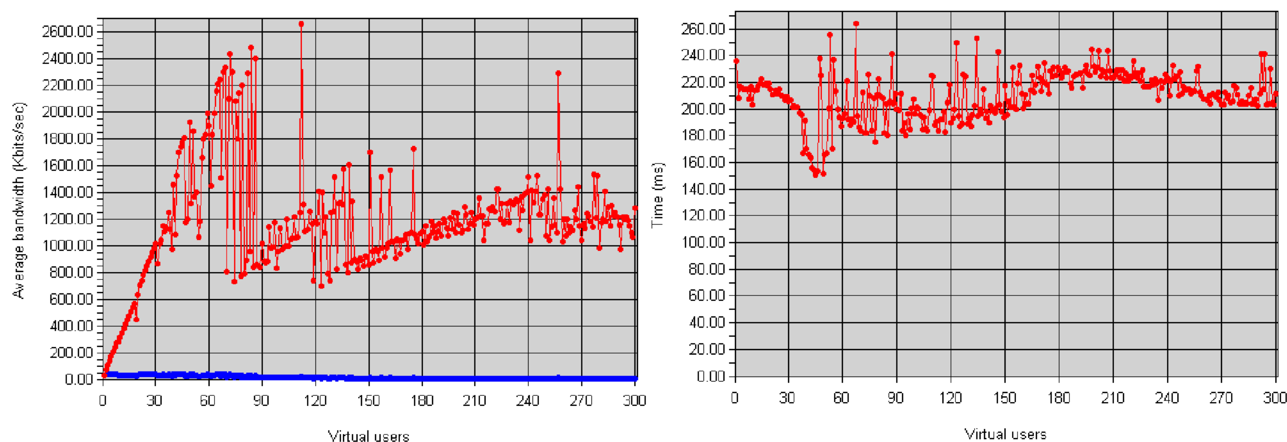


pav. 84 Testavimo rezultatai siunčiant 500 kilobaitų dydžio bylą su 1 milisekundės laukimo laiku.

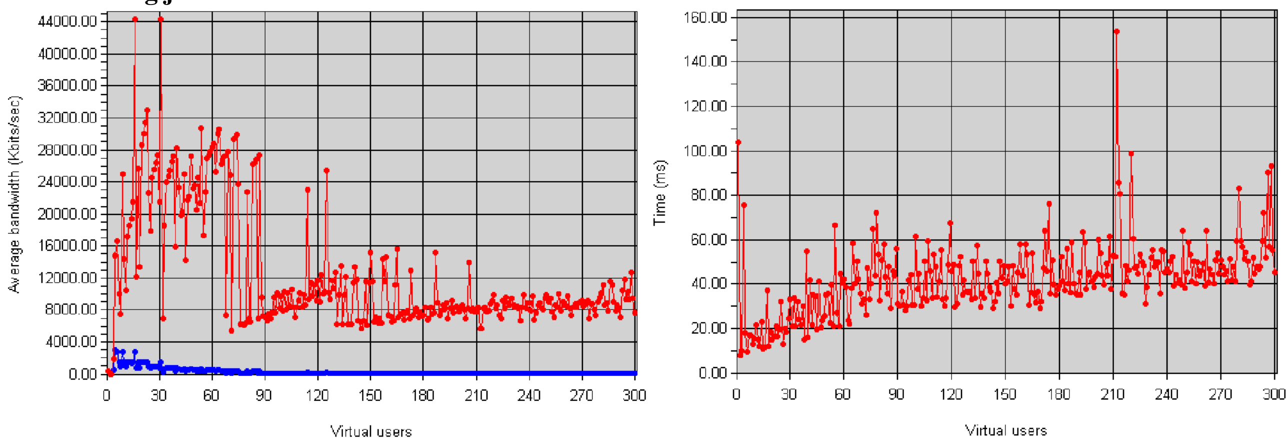
Šito modelio pateikti praktinio išbandymo rezultatai parodė, kad jis yra daug kartų stabilesnis klaidų atžvilgių negu visi prieš tai nagrinėjami modeliai realizuoti reaktoriaus schemos pagrindu. Testavimu metu buvo pastebėtas daugiausiai 1-3% visų bylų siuntimo klaidų procentas ir nepriklausė nuo to su kokiu nepertraukiamu laukimo laiku dirbo įvykių išskyrėjas. Toks rezultatas yra lengvai paaiškinamas prisijungimo eilės buvimų. Visos ryšio užmezgimo užklausos buvo kaupiamos ir iš eilės apdorojamos.

Didinant įvykio išskyrėjo laukimo laiką, kaip ir tikėtasi vidutinis užklausos apdorojimo laikas augo ir geriausią rezultatą parodė su vienos milisekundės laiku. Paliginus su kitų modelių vidutiniu transakcijos laiku pastebėtinas mažas sulėtėjimas.

3.5.2.5. Atskiros gijos ryšio užmezgėjui, gijos prisijungimo eilės apdorojimui ir parametrizuotas gijų skaičius su savais įvykių išskyrėjais užklausų apdorotojams modelis.

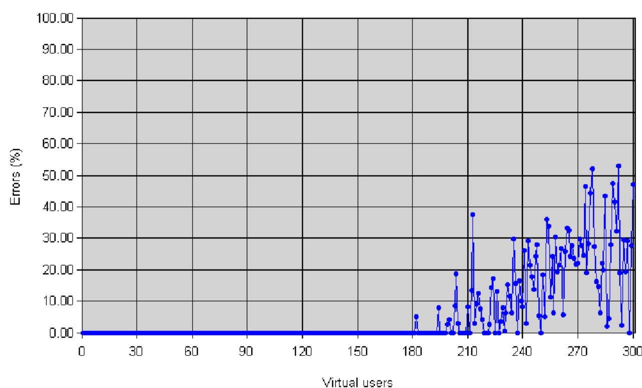


pav. 85 Testavimo rezultatai siunčiant 500 baitų dydžio bylą su 1 milisekundės laukimo laiku ir dviejomis darbinėmis gijomis.



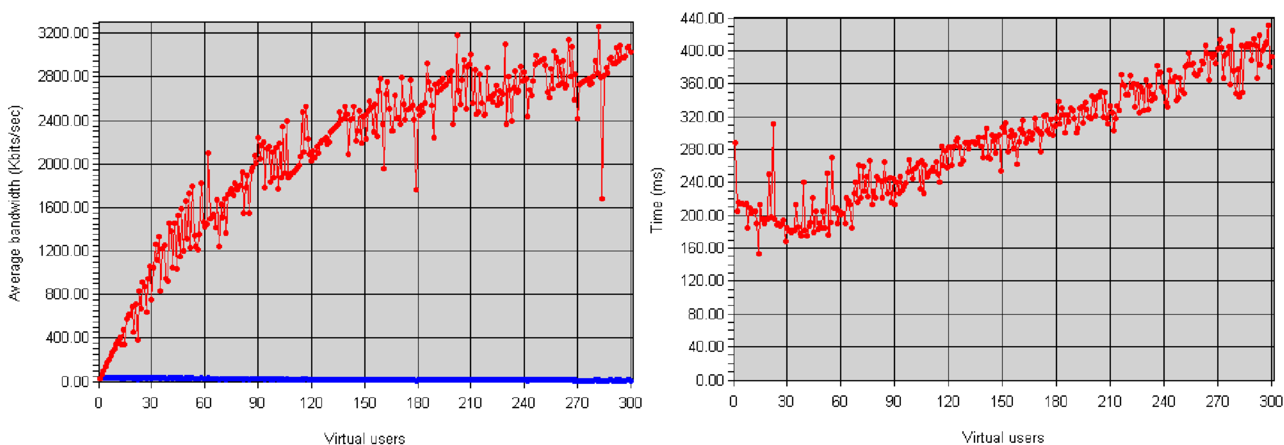
pav. 86 Testavimo rezultatai siunčiant 5 kilobaitų dydžio bylą su 1 milisekundės laukimo laiku ir dviejomis darbinėmis gijomis.

Tyrimo metu buvo pastebėta, kad nepriklausomai nuo parametrizuojamo šiam modeliui įvykio išskyrėjo laukimo laiko klaidų atsiradimo procentas buvo beveik stabilus siunčiant visų dydžių bylas. Kiekviename testavimo žingsnyje generuojamų klaidų grafikas atrodė taip kaip pavaizduota sekančiame paveikslėlyje pav. 87 su tam tikrais nežymiais nukrypimais. Tikėtina, kad padidinus apkrovimą klaidų kiekis dar labiau padidėtų. Dėl minėtos priežasties šio modelio testavimas buvo nutrauktas atlikus bandymus siunčiant 500 baitų ir 5 kilobaitų bylas.

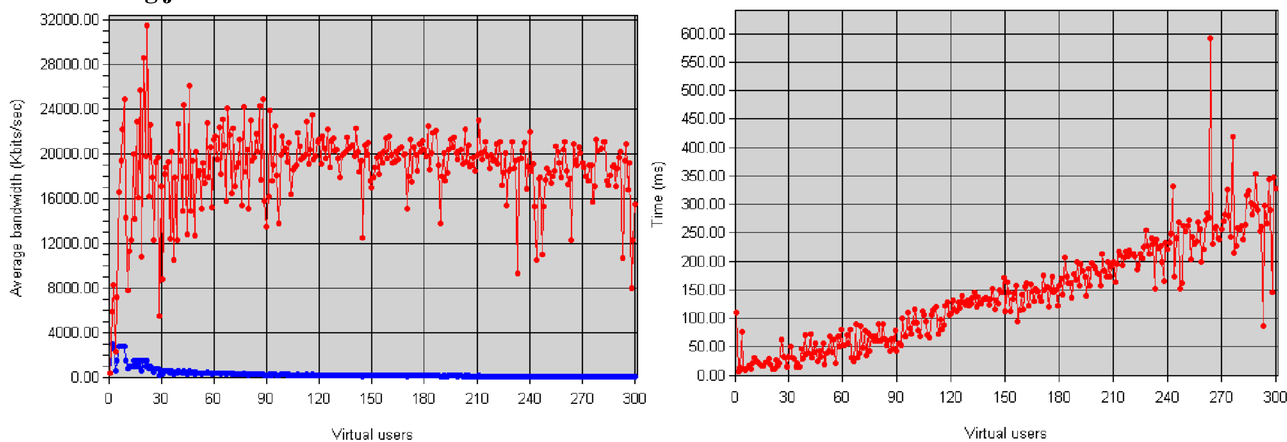


pav. 87 Klaidų kiekis siunčiant 5 kilobaitų dydžio bylą su 1 milisekundės laukimo laiku ir dviejomis darbinėmis gijomis.

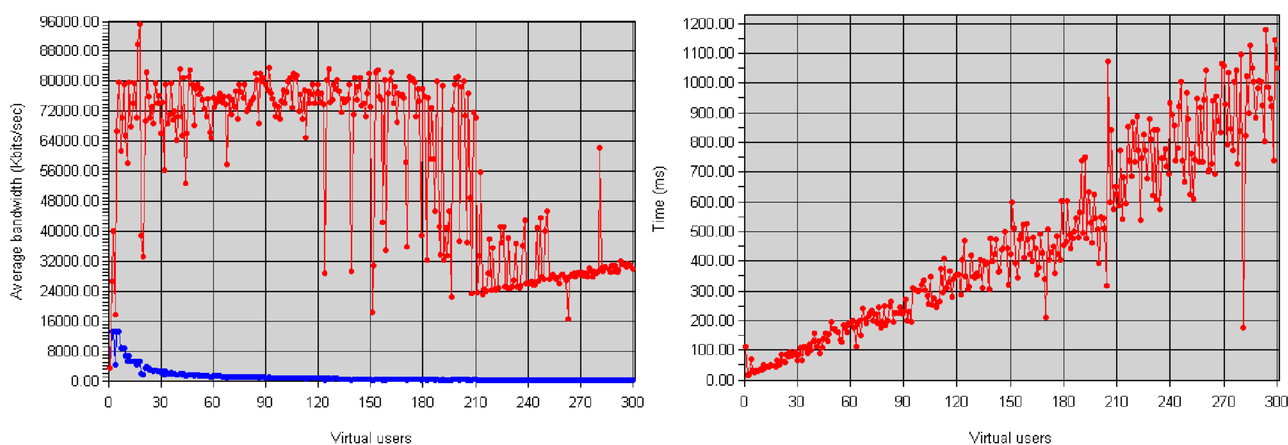
3.5.2.6. Atskiros gijos ryšio užmezgėjui, gijos prisijungimo eilės apdorojimui, gijos su įvykių išskyrėju užklausų apdorotojams bei parametrizuotas gijų skaičius skirtu užklausų vykdytojams modelis.



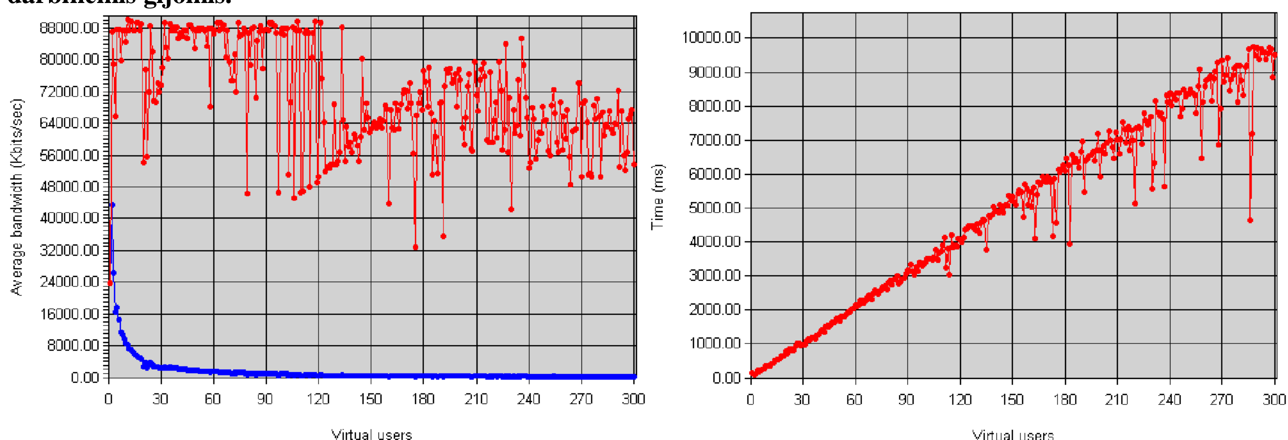
pav. 88 Testavimo rezultatai siunčiant 500 baitų dydžio bylą su 1 milisekundės laukimo laiku ir penkiomis darbinėmis gijomis.



pav. 89 Testavimo rezultatai siunčiant 5 kilobaitų dydžio bylą su 1 milisekundės laukimo laiku ir penkiomis darbinėmis gijomis.



pav. 90 Testavimo rezultatai siunčiant 50 kilobaitų dydžio bylą su 1 milisekundės laukimo laiku ir penkiomis darbinėmis gijomis.



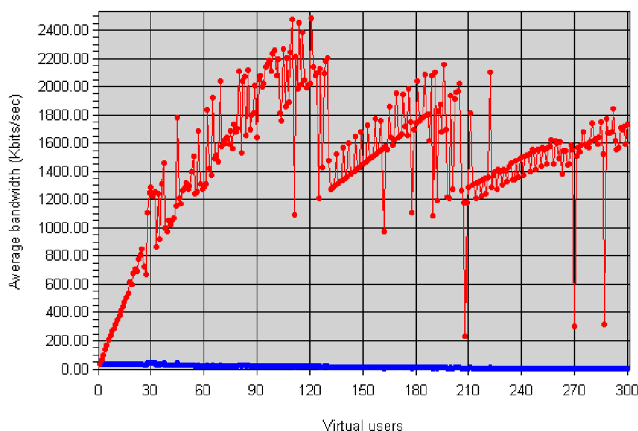
pav. 91 Testavimo rezultatai siunčiant 500 kilobaitų dydžio bylą su 1 milisekundės laukimo laiku ir penkiomis darbinėmis gijomis.

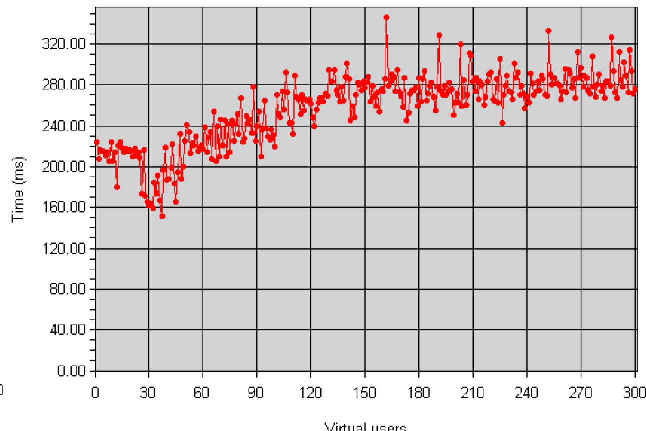
Testuojamas modelis buvo parametrizuojamas tokiais darbinių gijų skaičiais: 5, 50 ir 250. Atlikus testavimą ir paliginus gautus rezultatus geriausius matuojamus vienetus parodė modelis su penkiomis darbinėmis gijomis.

Šio modelio tyrimo metu kaip ir kito modelio naudojančio prisijungimo eilę klaidų kiekis yra vienas iš mažiausiu.

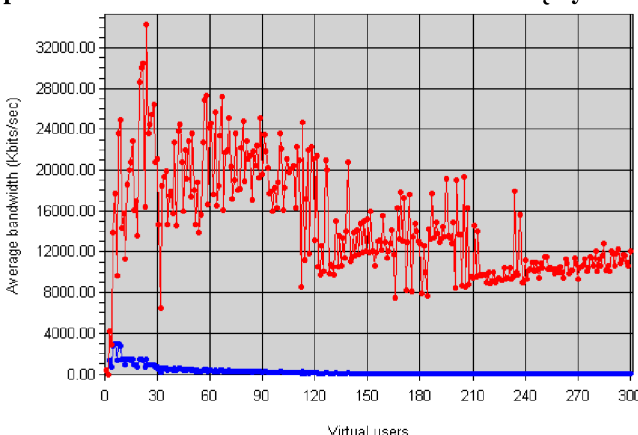
Realizuojant šį modelį buvo iškelta teorinė prielaida, kad sukūrus darbinių gijų grupę ir delegavę jai užklausų vykdymą bus pasiektas greitesnis vidutinis transakcijos laikas. Įvykdžius testavimo scenarijus buvo pastebėtas toks dalykas, kad modelis be darbinių gijų sparčiau veikė siunčiant mažesnio dydžio bylas, t.y. 500 baitų ir 5 kilobaitų, kai tuo tarpu šiame skyriuje testuojamo modelio vidutinis transakcijos laikas buvo mažesnis siunčiant 50 ir 500 kilobaitų dydžio bylas.

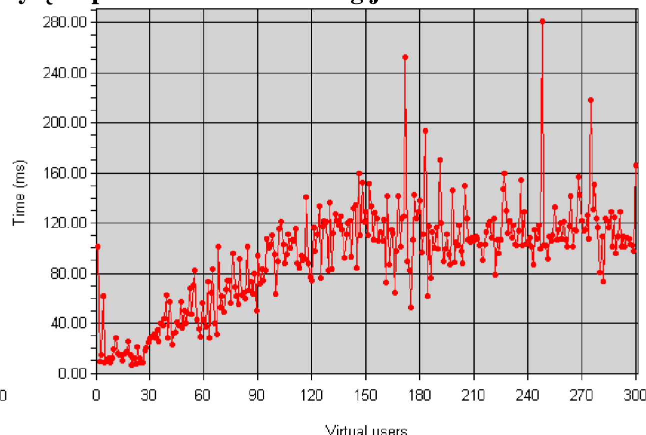
3.5.2.7. Vienos gijos ir vieno įvykių išskyrėjo modelis su parametrizuota gijų grupe skirta užklausų vykdytojams.



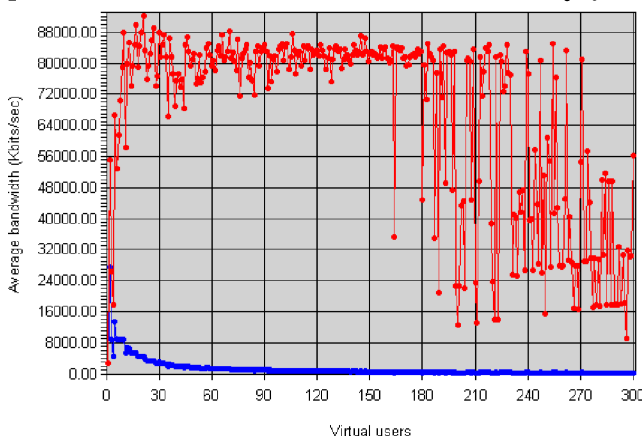


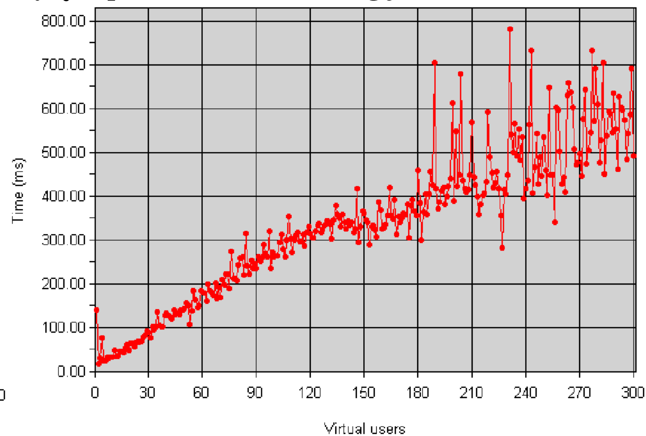
pav. 92 Testavimo rezultatai siunčiant 500 baitų dydžio bylą su penkiomis darbinėmis gijomis.



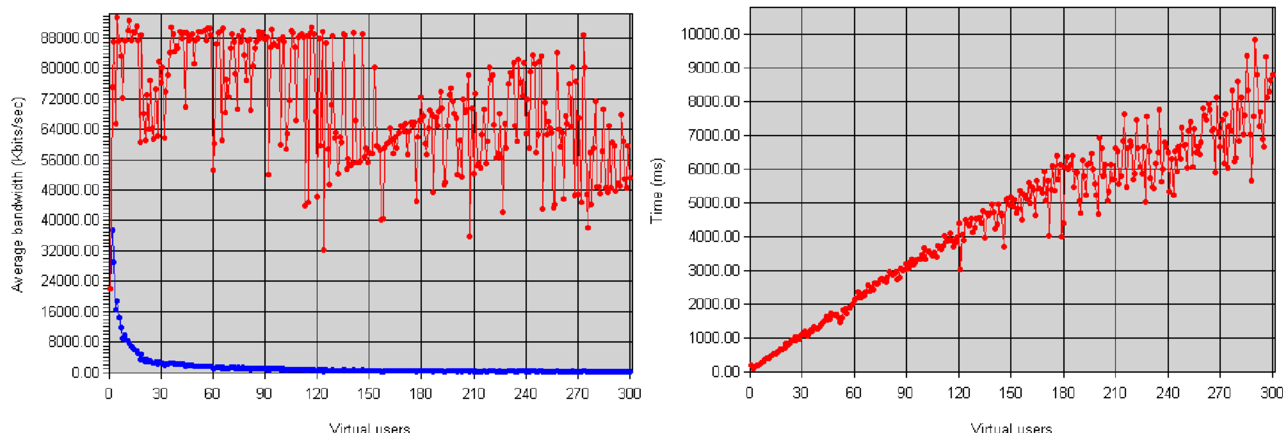


pav. 93 Testavimo rezultatai siunčiant 5 kilobaitų dydžio bylą su penkiomis darbinėmis gijomis.





pav. 94 Testavimo rezultatai siunčiant 50 kilobaitų dydžio bylą su penkiomis darbinėmis gijomis.



pav. 95 Testavimo rezultatai siunčiant 500 kilobaitų dydžio bylą su penkiomis darbinėmis gijomis.

Testuojamas modelis buvo parametrizuojamas tokiais darbinių gijų skaičiais: 5, 50 ir 250. Atlikus testavimą ir paliginus gautus rezultatus, geriausius matuojamus vienetus parodė modelis su penkiomis darbinėmis gijomis.

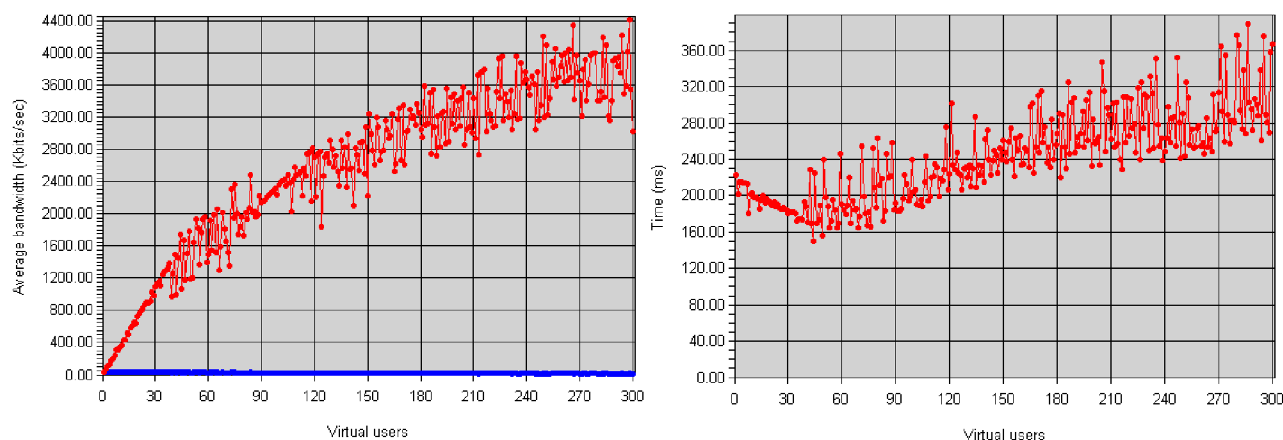
Priešingai negu modelio su prisijungimo eiles atveju, ši konkurentiškumą valdančio komponento modifikacija veikė prasčiau, generuodama apie 5-7% daugiau klaidų.

Realizuojant šį modelį buvo norėta praktiškai paliginti modelius su prisijungimo eile ir be jos panaudojant darbines gijas. Įvykdžius testavimo scenarijus buvo pastebėta, kad modelis be prisijungimo eilės veikė apie 60% greičiau negu modelis su prisijungimo eile.

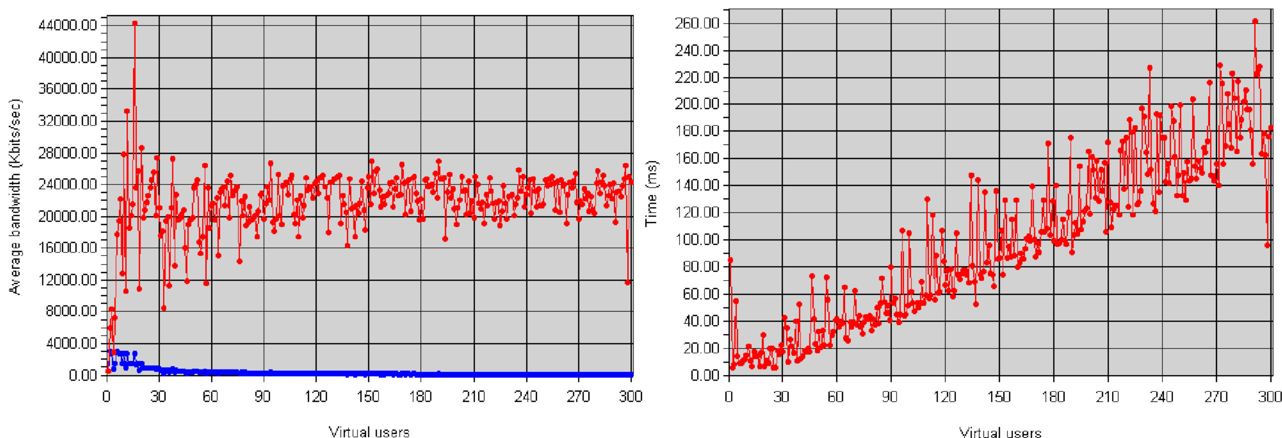
Palyginus vienos gijos su vienu įvykių išskyrėju modelio, bet be darbinių gijų rezultatus matomas nagrinėjamo modelio pranašumas siunčiant tik didesnio dydžio bylas.

3.5.3. Asinchroninis užklausų apdorojimas (Proaktoriaus schema)

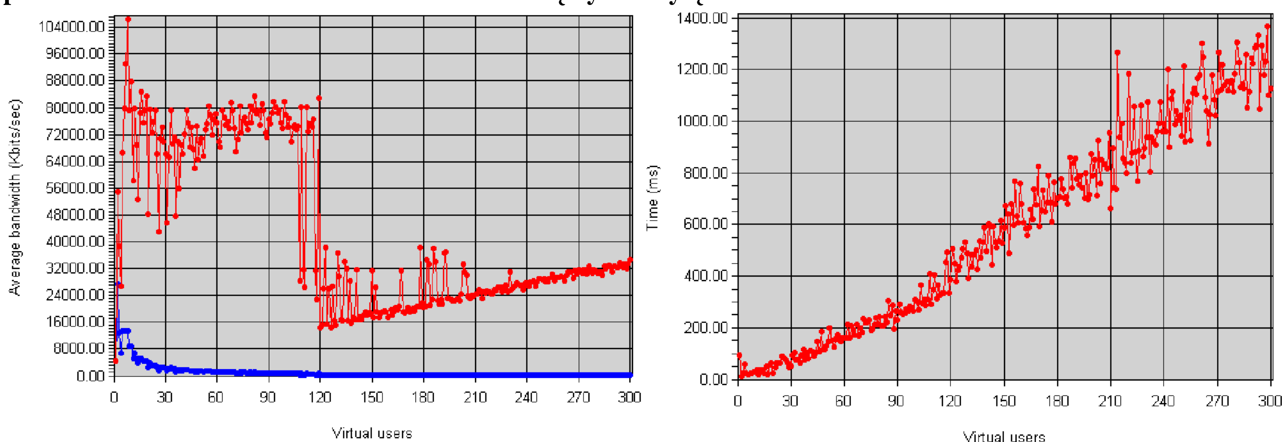
3.5.3.1. Vienos gijos ir grįžtamojo kvietimo modelis.



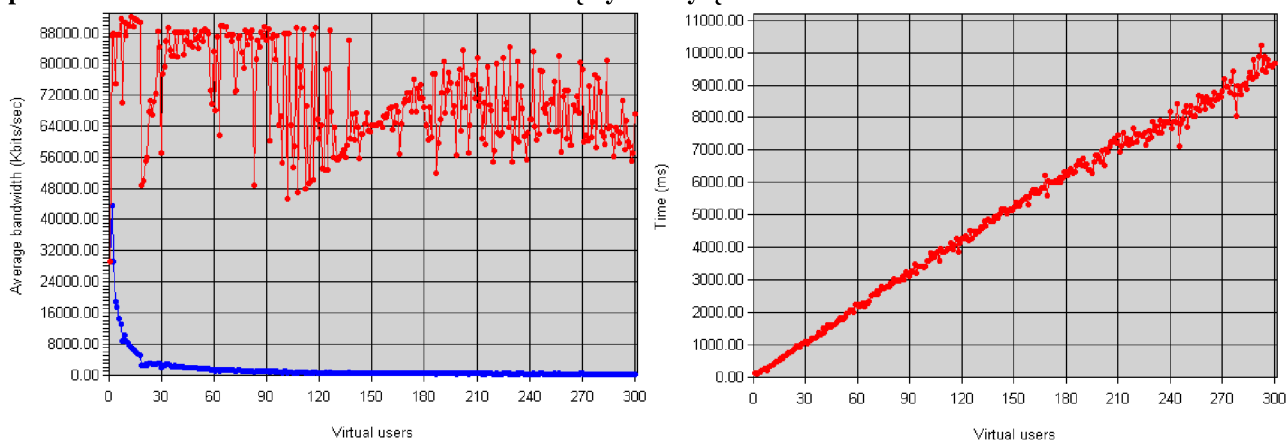
pav. 96 Testavimo rezultatai siunčiant 500 baitų dydžio bylą.



pav. 97 Testavimo rezultatai siunčiant 5 kilobaitų dydžio bylą.



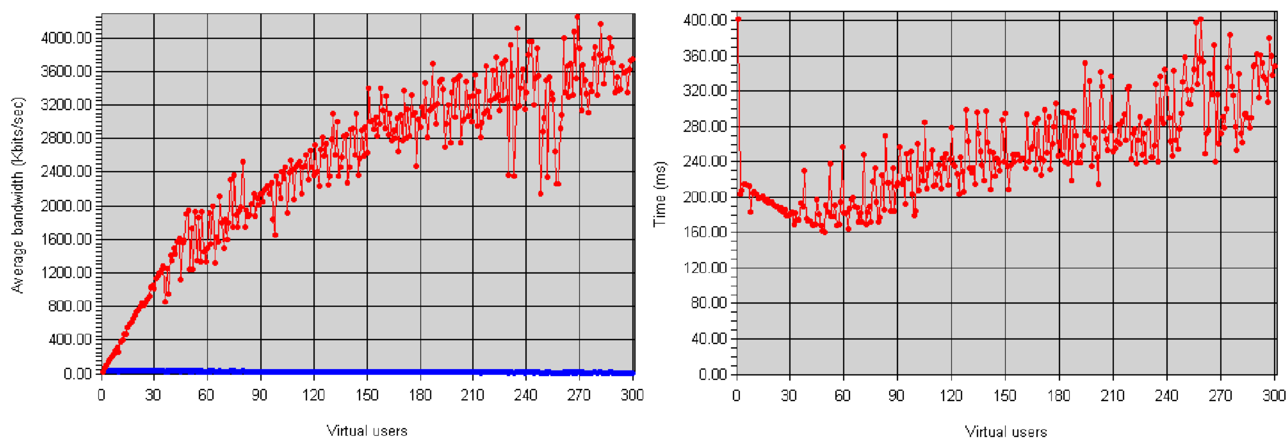
pav. 98 Testavimo rezultatai siunčiant 50 kilobaitų dydžio bylą.



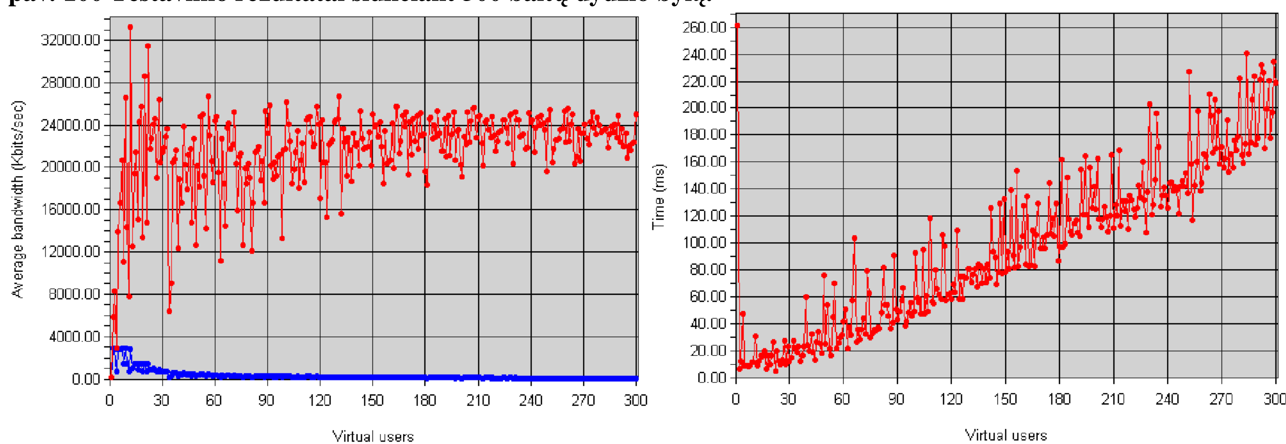
pav. 99 Testavimo rezultatai siunčiant 500 kilobaitų dydžio bylą.

Pateikti rezultatai iš esmės rodo tas pačias charakteristikas kaip ir vienos gijos vienam prisijungimui bei rektoriaus pagrindu realizuotu modelių rezultatai. Augant konkurentiškų klientų kiekiui bendrai paėmus vidutinis serverio pralaidumas krenta, o vidutinis transakcijos laikas auga. Tyrimo metu buvo pastebėtas mažas klaidų kiekis.

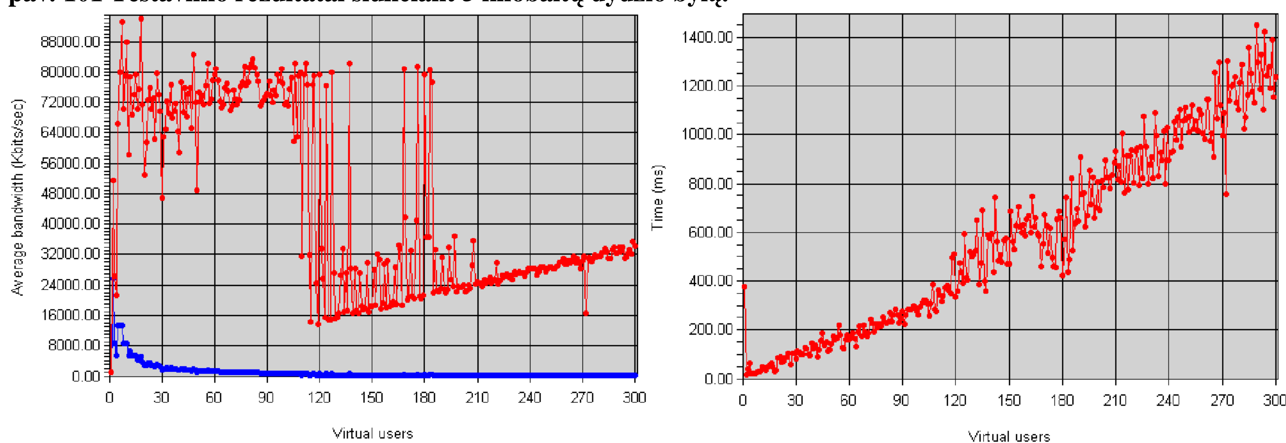
3.5.3.2. Atskiros gijos ryšio užmezgėjui, gijos prisijungimo eilės apdorojimui ir asinchroniškų užklausos apdorotojų su grįžtamuoju kvietimu modelis.



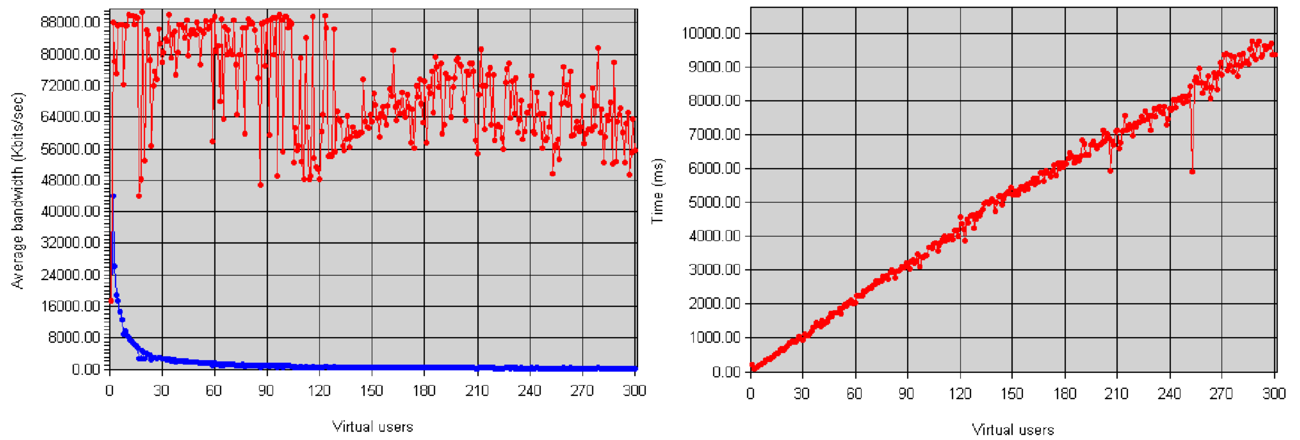
pav. 100 Testavimo rezultatai siunčiant 500 baitų dydžio bylą.



pav. 101 Testavimo rezultatai siunčiant 5 kilobaitų dydžio bylą.



pav. 102 Testavimo rezultatai siunčiant 50 kilobaitų dydžio bylą.



pav. 103 Testavimo rezultatai siunčiant 500 kilobaitų dydžio bylą.

Realizuojant šį modelį buvo iškeltas klausimas ar grįžtamojo kvietimo metodo užregistravimas asinchroninių operacijų kontroleryje įtakoja bendrą užklausų apdorojimo laiką. Palyginus pateiktus rezultatus matosi, kad greičių skirtumas nors ir nežymus yra šio modelio nenaudai. Prisijungimo eilės aptarnavimas užima daugiau laiko nei minėtas registravimo veiksmas. Iš kitos pusės aptinkamas klaidų kiekis teoriškai turėtų būti mažesnis, bet sprendžiant iš rezultatų klaidų atsiradimas ir paskirstymas vieno ir kito modelio atveju yra panašus.

3.5.4. Išteklių panaudojimas.

Žemiau pateiktuose lentelėse yra užfiksuoti procesoriaus ir atminties panaudojimai kiekvieno modelio testavimo metu aprašytų ankstesniuose skyreliuose.

Siunčiamų bylų dydžiai

Tyrimo rezultatų skyriai

	500B	5KB	50KB	500KB
3.5.1.1	41	33	46	40
3.5.1.2	45	48	45	46
3.5.1.3 su 5 gijomis	5	-	-	-
3.5.1.3 su 50 gijomis	22	40	19	25
3.5.1.3 su 250 gijomis	44	43	38	39
3.5.1.3 su 500 gijomis	41	-	-	-
3.5.2.1	36	43	50	50
3.5.2.2.1 su 0 sek. laukimu	99	99	99	99
3.5.2.2.1 su 1 sek. laukimu	40	41	50	45
3.5.2.2.1 su 50 sek. laukimu	34	18	37	-
3.5.2.2.1 su 250 sek. laukimu	38	-	-	-
3.5.2.2.1 su 500 sek. laukimu	40	-	-	-
3.5.2.2.2 su 0 sek. Laukimu	55	55	54	52
3.5.2.2.2 su 1 sek. laukimu	42	54	50	51
3.5.2.2.2 su 50 sek. laukimu	40	34	43	41
3.5.2.2.2 su 250 sek. laukimu	41	-	-	-
3.5.2.3 su 0 sek. Laukimu	53	56	53	-
3.5.2.3 su 1 sek. laukimu	43	41	50	45
3.5.2.3 su 50 sek. laukimu	42	29	50	-
3.5.2.3 su 250 sek. laukimu	40	-	-	-
3.5.2.4 su 0 sek. Laukimu	53	55	50	-
3.5.2.4 su 1 sek. laukimu	45	44	50	52
3.5.2.4 su 50 sek. laukimu	50	36	50	-
3.5.2.4 su 250 sek. laukimu	34	-	-	-
3.5.2.4 su 500 sek. laukimu	28	-	-	-
3.5.2.5 su 2 gijomis	79	80	-	-
3.5.2.5 su 5 gijomis	88	-	-	-
3.5.2.5 su 10 gijomis	91	-	-	-
3.5.2.6 su 5 gijomis	59	67	65	80
3.5.2.6 su 50 gijomis	61	66	64	-
3.5.2.6 su 250 gijomis	67	65	66	-
3.5.2.7 su 5 gijomis	54	53	51	53
3.5.2.7 su 50 gijomis	54	48	47	-
3.5.2.7 su 250 gijomis	60	59	58	-
3.5.3.1	39	36	28	23
3.5.3.2	34	35	23	22

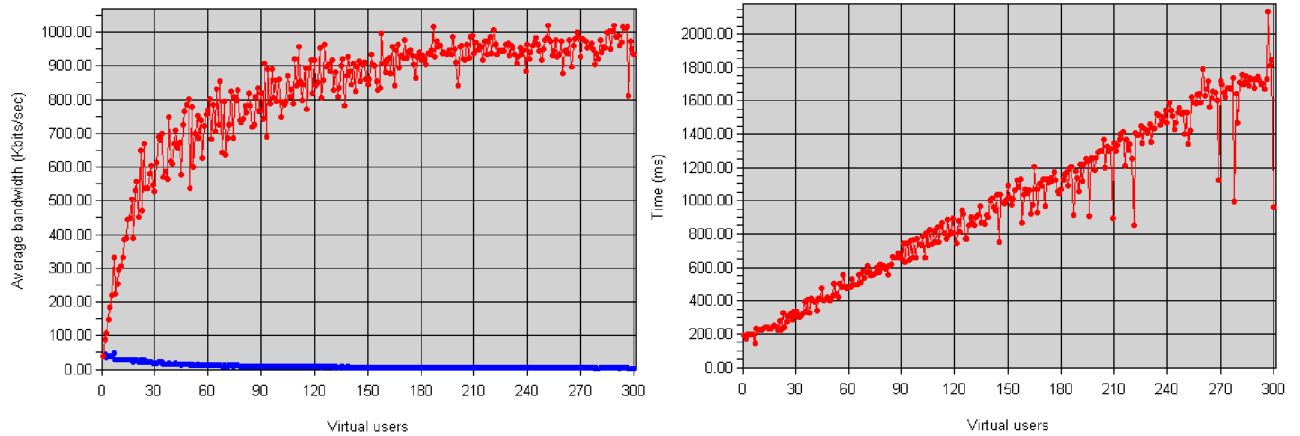
lentelė 2 Procesoriaus panaudojimas, procentais.

	500B	5KB	50KB	500KB
3.5.1.1	31,800	32,340	32,632	33,244
3.5.1.2	34,876	34,792	34,040	34,760
3.5.1.3 su 5 gijomis	12.332	-	-	-
3.5.1.3 su 50 gijomis	16,196	16,660	16,288	33,532
3.5.1.3 su 250 gijomis	31,992	33,204	33,448	16,516
3.5.1.3 su 500 gijomis	40,159	-	-	-
3.5.2.1	24,660	25,492	25,492	25,820
3.5.2.2.1 su 0 sek. laukimu	24,528	24,820	24,868	24,892
3.5.2.2.1 su 1 sek. laukimu	24,664	25,408	25,856	25,312
3.5.2.2.1 su 50 sek. laukimu	24,336	26,412	26,380	-
3.5.2.2.1 su 250 sek. laukimu	24,284	-	-	-
3.5.2.2.1 su 500 sek. laukimu	24,392	-	-	-
3.5.2.2.2 su 0 sek. Laukimu	24,328	24,744	25,188	25,749
3.5.2.2.2 su 1 sek. laukimu	24,608	25,372	25,740	25,872
3.5.2.2.2 su 50 sek. laukimu	24,324	26,420	26,432	25,484
3.5.2.2.2 su 250 sek. laukimu	24,780	-	-	-
3.5.2.3 su 0 sek. Laukimu	24,444	25,252	25,376	-
3.5.2.3 su 1 sek. laukimu	24,600	25,234	26,464	25,652
3.5.2.3 su 50 sek. laukimu	24,556	25,960	25,900	-
3.5.2.3 su 250 sek. laukimu	24,452	-	-	-
3.5.2.4 su 0 sek. Laukimu	24,736	24,840	25,196	-
3.5.2.4 su 1 sek. laukimu	24,524	25,120	25,716	25,416
3.5.2.4 su 50 sek. laukimu	24,196	24,644	24,096	-
3.5.2.4 su 250 sek. laukimu	24,260	-	-	-
3.5.2.4 su 500 sek. laukimu	24,144	-	-	-
3.5.2.5 su 2 gijomis	23,800	24,296	-	-
3.5.2.5 su 5 gijomis	19,954	-	-	-
3.5.2.5 su 10 gijomis	16,520	-	-	-
3.5.2.6 su 5 gijomis	17,560	17,600	17,576	18,620
3.5.2.6 su 50 gijomis	20,860	20,736	20,828	-
3.5.2.6 su 250 gijomis	36,184	35,572	38,264	-
3.5.2.7 su 5 gijomis	16,328	16,288	17,384	18,576
3.5.2.7 su 50 gijomis	19,408	18,660	18,120	-
3.5.2.7 su 250 gijomis	31,244	35,988	36,468	-
3.5.3.1	19,548	19,052	18,868	18,524
3.5.3.2	17,940	18,050	19,576	18,744

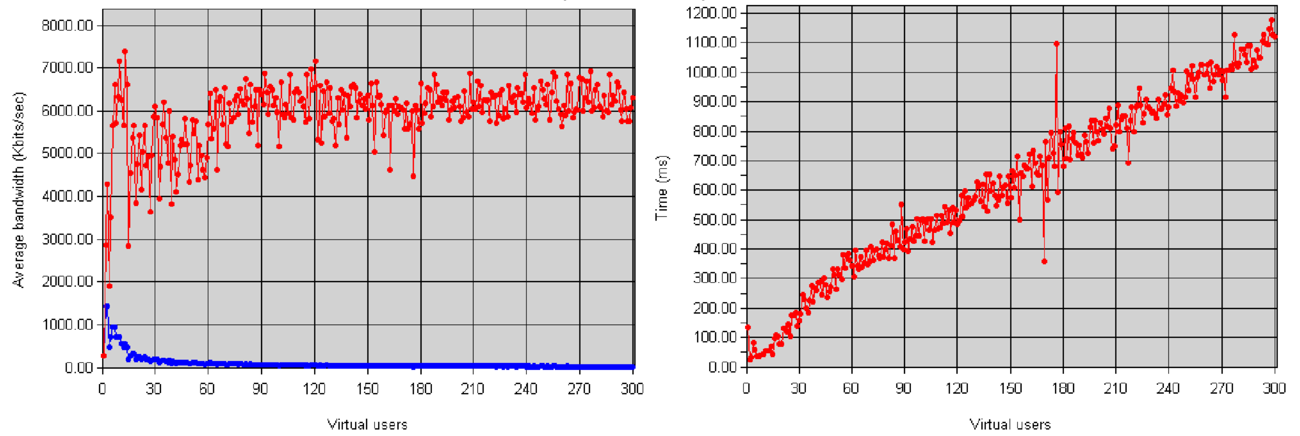
lentelė 3 Atminties sunaudojimas, kilobaitais

3.6. Trečios tyrimo dalies rezultatai.

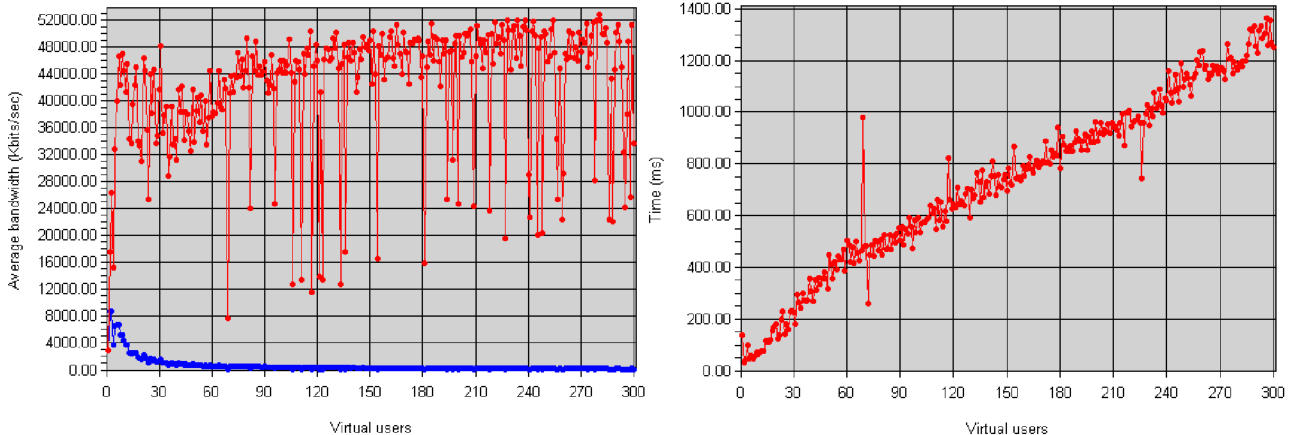
3.6.1. Vienos gijos vienam prisijungimui modelis su prisijungimo eile ir parametrizuota sukurtu gijų grupe.



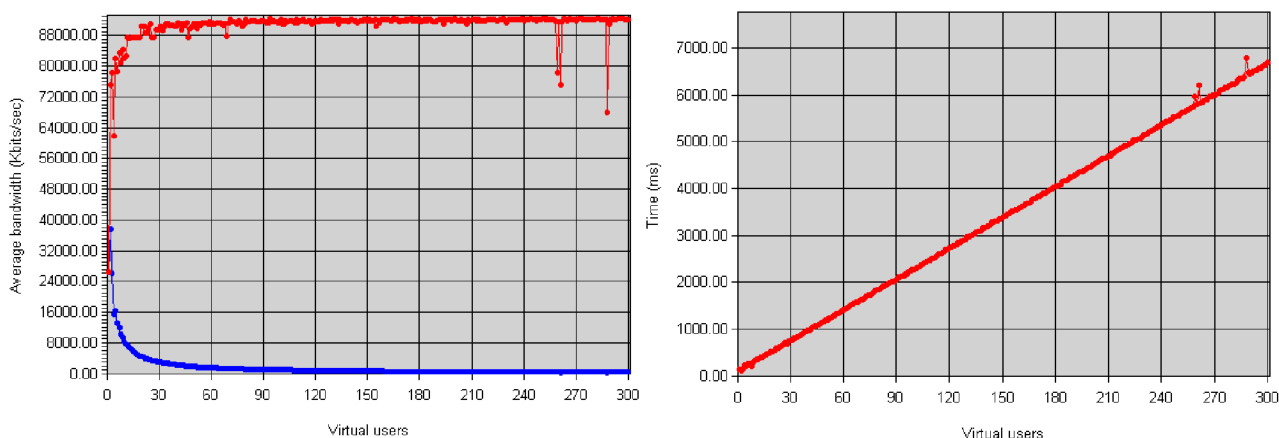
pav. 104 Testavimo rezultatai siunčiant 500 baitų dydžio bylą su 250 darbinėmis gijomis.



pav. 105 Testavimo rezultatai siunčiant 5 kilobaitų dydžio bylą su 50 darbinėmis gijomis.



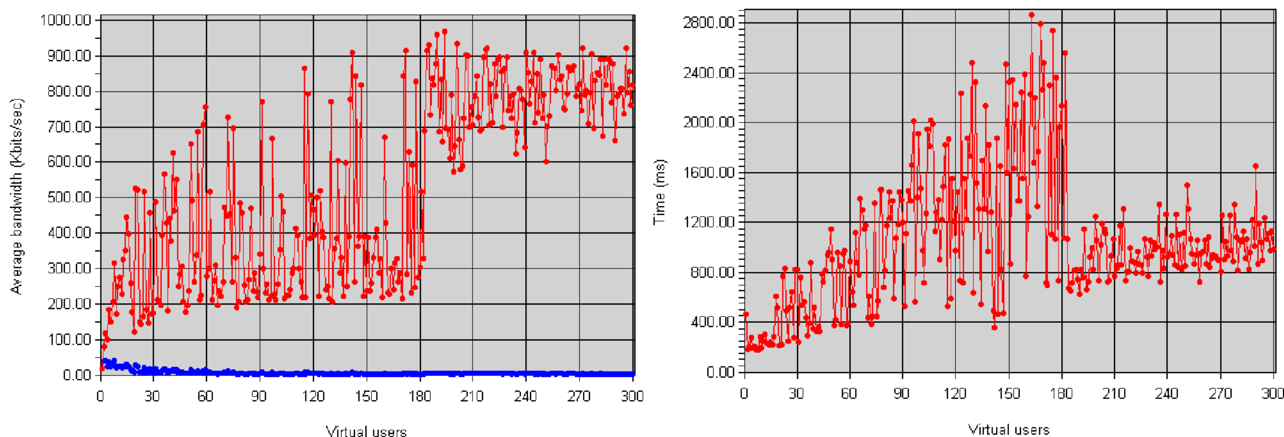
pav. 106 Testavimo rezultatai siunčiant 50 kilobaitų dydžio bylą su 50 darbinėmis gijomis.



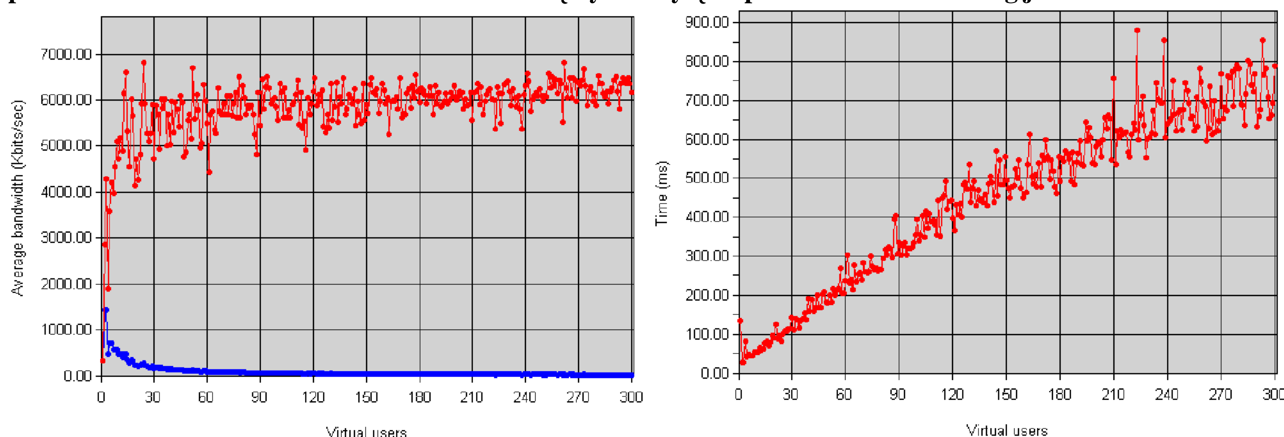
pav. 107 Testavimo rezultatai siunčiant 500 kilobaitų dydžio bylą su 5 darbinėmis gijomis.

Pateikti aukščiau rezultatai parodo konkurentiškumą valdančio modelio veikimą esant papildomas darbui. Palyginus su to pačio modelio rezultatais, bet be papildomo darbo matome, kad vidutinio transakcijos laiko kreivė auga greičiau. Tuo tarpu vidutinis pralaidumas išlieka panašus, su momentiniais, kaip rodo pav. 106 paveikslas, pralaidumo kritimais.

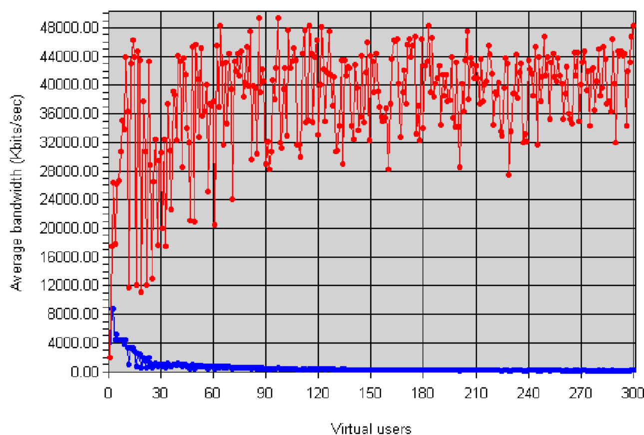
3.6.2. Vienos gijos ir vieno įvykių išskyrėjo modelis su parametrizuota gijų grupe skirta užklausų vykdytojams.

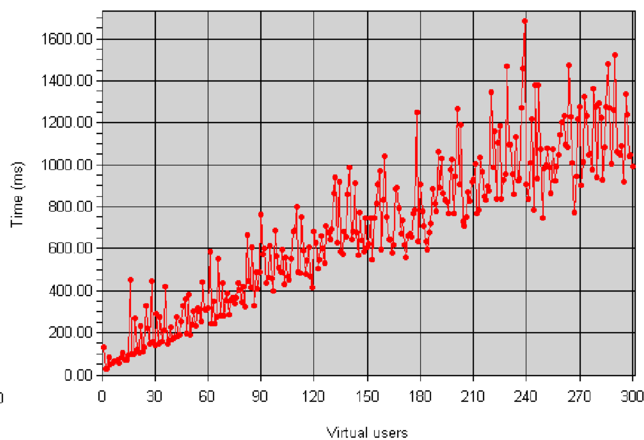


pav. 108 Testavimo rezultatai siunčiant 500 baitų dydžio bylą su penkiomis darbinėmis gijomis.

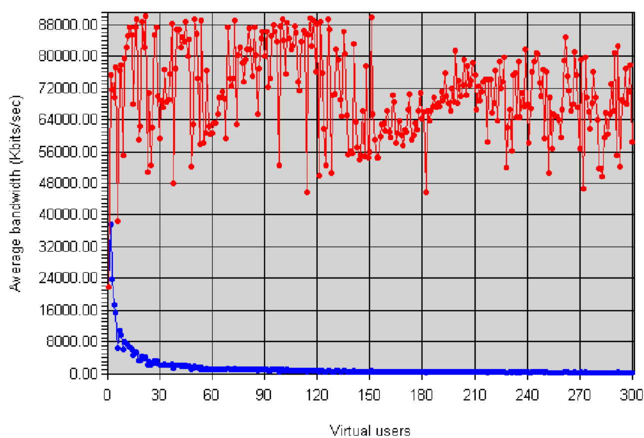


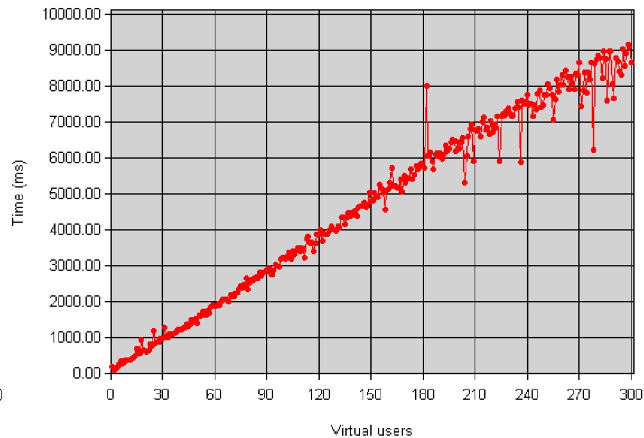
pav. 109 Testavimo rezultatai siunčiant 5 kilobaitų dydžio bylą su penkiomis darbinėmis gijomis.





pav. 110 Testavimo rezultatai siunčiant 50 kilobaitų dydžio bylą su penkiomis darbinėmis gijomis.

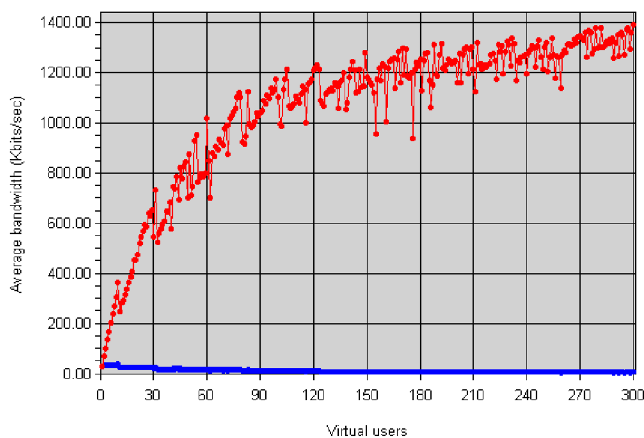


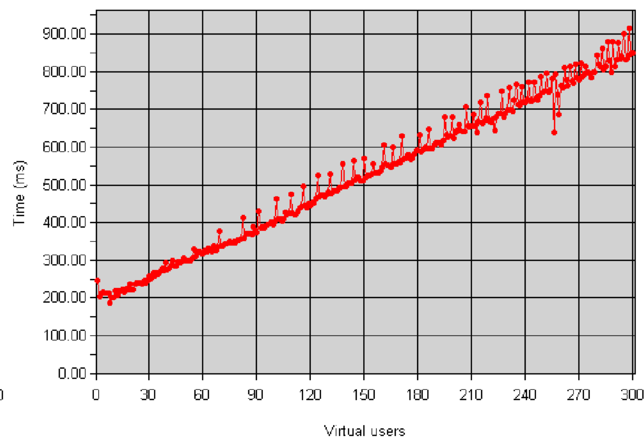


pav. 111 Testavimo rezultatai siunčiant 500 kilobaitų dydžio bylą su penkiomis darbinėmis gijomis.

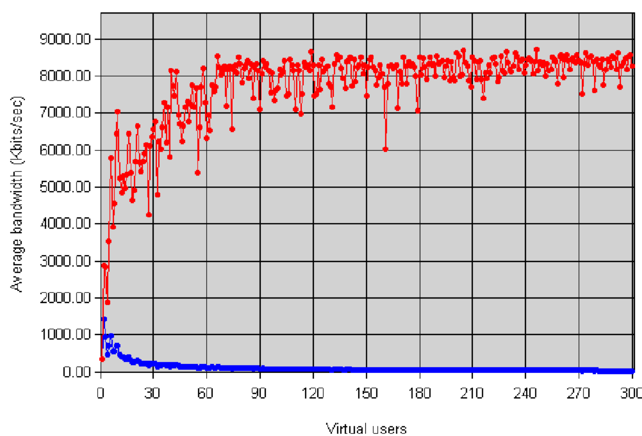
Pateikti aukščiau testavimo rezultatai, kaip ir pateikti praeitame skyrelyje turi tą pačią tendenciją, t.y. vidutinio transakcijos laiko kreivė didėja sparčiau negu modelio be papildomo darbo atveju. Tuo tarpu vidutinis pralaidumas lyginant modelio be papildomo darbo yra žymiai stabilesnis.

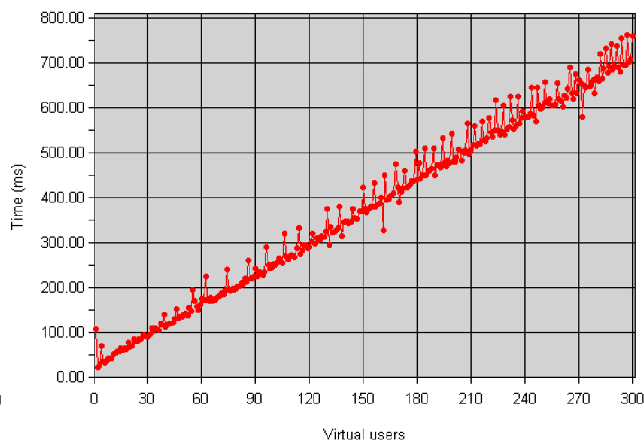
3.6.3. Vienos gijos ir grįžtamojo kvietimo modelis.



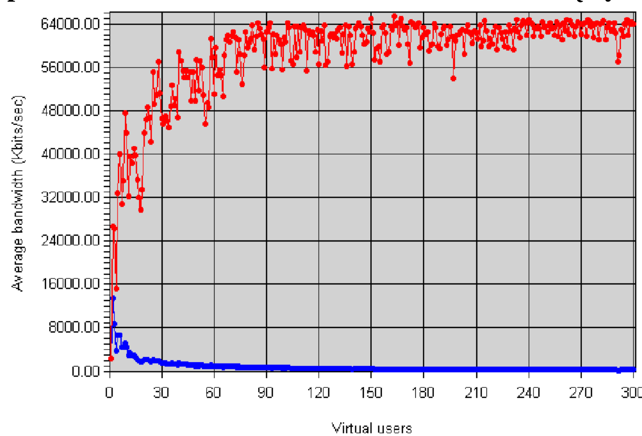


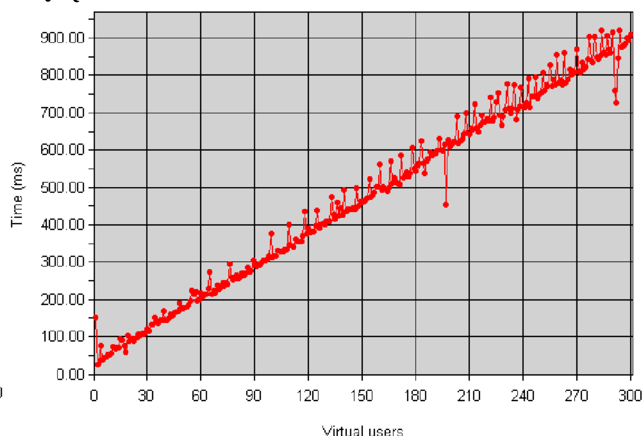
pav. 112 Testavimo rezultatai siunčiant 500 baitų dydžio bylą.



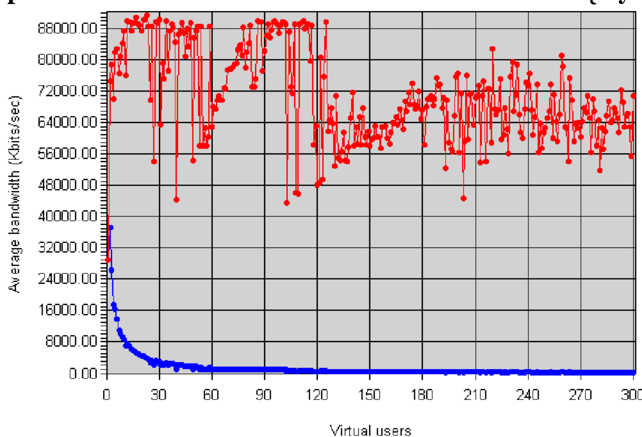


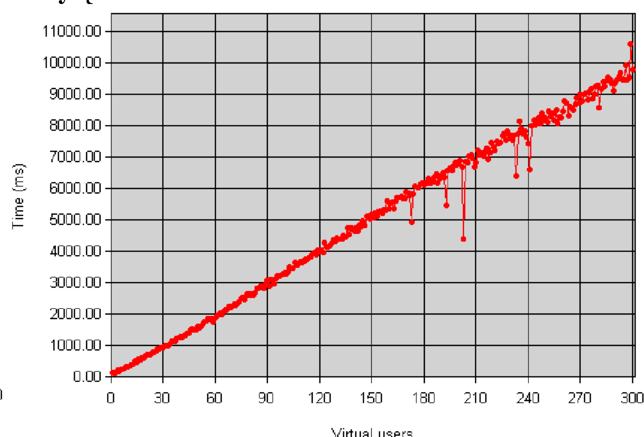
pav. 113 Testavimo rezultatai siunčiant 5 kilobaitų dydžio bylą.





pav. 114 Testavimo rezultatai siunčiant 50 kilobaitų dydžio bylą.





pav. 115 Testavimo rezultatai siunčiant 500 kilobaitų dydžio bylą.

Pateikti aukščiau testavimo rezultatai turi analogiška tendencija kaip ir pateikti praeitame skyrelyje. Vidutinio transakcijos laiko kreivė didėja greičiau, o vidutinis pralaidumas lyginant modelio be papildomo darbo yra stabilesnis.

Išvados

Šio darbo tikslas buvo teoriškai ir praktiškai išnagrinėti kaip ir kokius žinomus užklausų srautų aptarnavimo sprendimo būdus galima realizuoti JAVA programavimo kalbos priemonėmis. Analizuojant programines JAVA kalbos galimybes didesnis dėmesys buvo atkreiptas į sąlyginai neseniai pasirodžiusį nesiblokuojančio įvedimo ir išvedimo posistemį (NIO). Tuo tarpu ieškodamas programavimo kalboje asinchroninio skaitymo ir rašymo operacijų vykdymo buvo išnagrinėta IBM kompanijos sukurta asinchroninio įvedimo ir išvedimo biblioteka (AIO4J) realizuota panaudojant JNI sąsają.

Darbo metu buvo nagrinėjami dviejų tipų konkurentiškumą valdantys modeliai, t.y. gijos pagrindu bei įvykių iškyrimo ir reagavimo pagrindu realizuoti posistemiai. Išanalizavus JAVA programavimo kalbos galimybes, paaiškėjo, kad visus modelius galima kuo puikiausiai sumodeliuoti ir realizuoti panaudojant sinchroninį besiblokuojantį įvedimo ir išvedimo posistemį, sinchroninį nesiblokuojantį įvedimo ir išvedimo posistemį (NIO) ir asinchroninį įvedimo ir išvedimo posistemį (AIO4J).

Darbo metu atliktas tyrimas buvo labiau orientuojamas į užklausų srauto apdorojančio komponento darbo analizę. Todėl sukurtas WEB serveris buvo suprojektuotas taip, kad užklausų vykdymas turėtų kaip galima mažesnę įtaką gautiems rezultatams. Nors įprastomis WEB serverio veikimo sąlygomis, užklausų vykdymas turėtų įtakoti užklausų srautą apdorojantį komponentą, vienos tyrimo dalies metu atliktas testavimas su dirbtinai apsunkintu užklausų vykdymu parodė, kad iš esmės rezultatai buvo labai panašūs.

Atlikus praktinį konkurentiškumą valdančių realizacijų tyrimą paaiškėjo tokie faktai:

- Visi realizuoti modeliai turi vieną bendrą charakteristiką, t.y. pastebima pralaidumo kritimo bei transakcijos laiko augimo tendencija augant konkurentiškai veikiančių naudotojų kiekiui.
- Bazinio gijų pagrindu realizuoto konkurentiškumo modelio našumas krenta didėjant užklausų kiekiui, o pasiekus sukuriamų gijų ribą pradeda tiesiog trukti atminties išteklių.
- Bazinė reaktoriaus schemos realizacija, naudojanti nesiblokuojantį įvedimo ir išvedimo posistemį, lyginant su gijos pagrindu realizuotų konkurentiškumo apdorojimu, esant labai dideliam apkrovimui panaudoja dvigubai mažiau procesoriaus laiko, penkis kartus mažiau sunaudoja pagrindinės atminties, o našumas vidutiniškai viso tyrimo metu skyrėsi vos daugiau nei per pusę.
- Bazinė proaktoriaus schemos realizacija našumo atžvilgiu šiek tiek aplenkia reaktoriaus schemą esant labai dideliam apkrovimui.
- Palyginus su bazinės gijos pagrindu realizuotų konkurentiškumo apdorojimu kai esant ribiniam gijų skaičiui pradeda tiesiog trukti skaičiavimo resursų, bazinės reaktoriaus ir

proaktoriaus schemų realizacijos išlieka darbingos per ilgą testavimo laiką esant labai dideliam apkrovimui.

- Gijos pagrindu realizuotas modelis nepriklausomai ar yra naudojama prisijungimo eilė, ar ne yra daugiau tinkamas ilgai trunkančioms transakcijoms negu trumpoms, dėl to kad trumpos operacijos iššaukia didelį gijų skaičiaus sukūrimą per trumpą laiko tarpą ir konteksto keitimo tarp tų gijų.
- Parametrizuotos darbinių gijų grupės naudojimas praktiškai visuose naudojančias ją modeliuose yra pastebimas toks dalykas, kad siunčiant skirtingo dydžio bylas įtakos turi darbinių gijų skaičius. Siunčiant mažesnio dydžio bylas turi būti didesnis darbinių gijų skaičius ir atvirkščiai siunčiant didesnio dydžio bylas reikalingas mažesnis darbinių gijų skaičius
- Parametrizuoto naudojamų įvykių išskyrėjų skaičiaus modelis neduoda laukiamų teorinių pranašumų. Kuo daugiau yra naudojama įvykių išskyrėjų tuo didesnis tampa serverio išteklių poreikis ir atsirandančių klaidų kiekis.
- Reaktoriaus pagrindu realizuotas modelis geriau veikia siunčiant mažesnius failus, negu didelius, t.y. sparčiau veikia esant trumpai trunkančioms transakcijoms negu ilgai trunkančioms. Tai galima paaiškinti darbo metu pastebėta NIO posistemio savybe, kuri mano sukurto komponento atveju gali būti trukumu. Esmė yra ta, kad įvykių išskyrėjas gražindamas skaitymo įvykį negarantuoja, kad nuskaityta iš prisijungimo kanalo informacija yra pilna. Tai savo ruožtu apsunkina pačio modelio realizaciją, nes reikalingas tampa daugkartinis kreipimasis į įvykių išskyrėją, darbinės gijos paėmimas, skaitymo operacijos delegavimas ir pakartotinis jos vykdymas.
- Proaktoriaus schemos pagrindu realizuotas modelio darbas esant vidutiniam apkrovimui nepasireiškia jokias pranašumais palyginus su gijos ar įvykio išskyrėjo pagrindu realizuotais komponentais.

Darbe pateikti apkrovos ir našumo testavimo rezultatai parodo, kad vienokio ar kitokio modelio realizacijos kiekvienoje situacijoje nėra šimtu procentu efektyvios ir visada atliekamos be klaidų. Vienas modelis yra pranašesnis už kitą esant skirtingam apkrovos lygiui. Todėl žemiau aprašytos rekomendacijos paremtos darbu metu atliktais praktiniais modelių išbandymais turėtų pagelbėti kuriant naują arba optimizuojant egzistuojantį komponentą atsakinga už užklausų srautų aptarnavimą:

- Negalima sukurti konkurentiškų užklausų srauto aptarnaujančio komponento naudojančio tik vieną ar kitą konkretų modelį.

- Konkurentiškų užklausų srautą aptarnaujantis komponentas turi būti statiškai adaptuotas prie būsimo apkrovimo lygio ir užklausų apdorojimo pobūdžio pritaikydamas atitinkamą konkurentiškumo ir įvykių išskyrimo mechanizmą.
- Esant trumpai trunkančioms operacijoms, kaip pavyzdžiui nedidelio dydžio failo siuntimui rekomenduojama naudoti reaktoriaus pagrindu realizuotą modelį su parametrizuota darbinių gijų grupe.
- Esant ilgai trunkančioms operacijoms, kaip pavyzdžiui didelio dydžio failo siuntimui rekomenduojama naudoti gijos pagrindu realizuota modelį su prisijungimo eile ir parametrizuota darbinių gijų grupe.
- Norėdami sumažinti atsirandančių ryšio užmezgimo klaidų kiekį iki minimumo rekomenduojama naudoti prisijungimo eilę pildančio ryšio užmezgėjo modelį su atskira gija aptarnaujančia minėtą eilę.

Summary

The purpose of this master thesis is to analyze how many of known solutions for request stream handling can be implemented in JAVA programming language. Also, in addition to it, this master thesis analyzes the way those solutions can be implemented using JAVA programming language. There are defined tasks to investigate in the scope of this thesis, which include following: exploration of well known concurrency managing models; analysis of their implementation possibilities in JAVA by designing and creating real working components; their implementation workflow research under various stress loaded conditions and, finally, supply recommendations for optimal parameterization for mentioned component to gain the best possible balance of resource utilization, throughput and latency.

There were analyzed basically two types of concurrency managing models in this work, i.e. thread based and two models based on event notification principle, designed using Douglas Schmidt reactor and proactor design patterns.

During exploration of JAVA programming language possibilities to implement above mentioned models a special attention was paid to the new input/output system (NIO) which provides synchronous non-blocking input/output operations, whereas to process operations asynchronously there was explored asynchronous execution of input/output operations by using JAVA non-native external AIO4J library created by IBM corporation.

In order to perform stress and performance benchmarks for created component that handles stream of requests, a simple Web server was deployed as a component responsible for establishing connections and handling requests. By executing various performance and stress loading scripts on basic and their modified request handling components it was empirically showed that in order to develop high quality component and to achieve optimal performance for it – a server must support static and partially dynamic adaptability features, i.e. possibility to parameterize server by various factors to adjust to probable loading level.

Literatūros sąrašas

[ALP04] IBM, AlphaWorks. *Asynchronous IO for Java.* http://www.alphaworks.ibm.com/tech/aio4j

[BOR05] Andrew Borg. Java NIO. The university of York, 2005. 37 psl.

[DF99] Alexandre Delarue, Eduardo B. Fernandez. Extension and Java Implementation of the Reactor-Acceptor-Connector Pattern Combination. 1999. 11 psl.

[DOU95] Douglas C. Schmidt. Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching. Addison-Wesley, 1995. 11 psl.

[DOU95] Douglas C. Schmidt. Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching. Addison-Wesley, 1995. 11 psl.

[EE04] Mike Edwards; Tim Ellison. Java Programming: The Java Async IO Package, 2004. http://java.sys-con.com/read/46658.htm

[HIT02] Ron Hitchens. Java Nio. O'Reilly, 312 psl, 2002

[LEA03] Doug Lea. Scalable IO in Java. State University of New York at Oswego, 2005. 39 psl.

[NEF03] John Neffenger. The Volano Report. 2003, 15 psl.

[PHC97+] Irfan Pyarali, Tim Harrison, Douglas C. Schmidt, and Thomas Jordan. Proactor: an Object Behavioral Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events. Illinois, September 1997. 14 psl.

[SOF06] SoftLogica, Web Application Testing. http://www.loadtestingtool.com

[SSH00+] Douglas C. Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann. Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects. John Willey & Sons, West Sussex, 482 psl, 2000.

[SUN02] Sun Microsystems, Inc. New I/O APIs. http://java.sun.com/j2se/1.4.2/docs/guide/nio/ , 2002

[VIS05] Krishnan Viswanath. Java New Input/Output. http://java.sys-con.com/read/43555.htm

[WCB01] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. Canada, 2001. 14 psl.

[WEL02] Matthew David Welsh. An Architecture for Highly Concurrent, Well-Conditioned Internet Services. Phd thesis, University of California at Berkeley, 2002. 211 psl.

[WGB03+] Matt Welsh, Steven D. Gribble, Eric A. Brewer, and David Culler. A Design Framework for Highly Concurrent Systems. University of California, Berkeley, 2003. 14 psl.