VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
PROGRAMŲ SISTEMŲ KATEDRA

Architektūrinių stilių, taikomų išskirstytoms tinkle programų sistemoms, tyrimas naudojant agentais grįstą modeliavimą

Architectural styles, applied for a distributed over a network software, research using agent-based modelling

Baigiamasis magistro darbas

Atliko:	2 kurso 2 grupės studentas Artūras Jankus	(parašas)
Darbo vadovas:	doc. dr. Karolis Petrauskas	(parašas)
Recenzentas:	doc. dr. Audronė Lupeikienė	(parašas)

Vilnius, 2018

Padėka

Darbo autorius dėkoja magistrinio darbo vadovui dr. Karoliui Petrauskui už pastabas, pasiūlymus, pastebėjimus bei nuolatinę pagalba magistrinio darbo rengimo laikotarpiu.

Santrauka

Šiame darbe kuriamas ir pateikiamas metodas, kuris leidžia tirti ir nustatinėti išskirstytų tinkle programų sistemų architektūrinių sprendimų ir stilių nefunkcines savybes, kurias jie suteikia programų sistemoms. Darbo pradžioje analizuojamos programų sistemų savybės, kuriomis jos gali pasižymėti, analizuojami architektūriniai stiliai. Apžvelgiami esami metodai bei metodikos, leidžiančios vertinti programų sistemų savybes. Darbo pabaigoje pateikiami sukurtojo metodo pritaikymo pavyzdžiai.

Raktiniai žodžiai: programų sistemų architektūra, architektūriniai stiliai, agentais grįstas modeliavimas, simuliacija, nefunkcinės savybės, išskirstymas tinkle, PACELC teorema, CAP teorema.

Summary

Method is created in this master thesis, that allows investigation and assessment of distributed over a network software systems non-functional qualities, that are brought by architectural decisions and styles. In the beginning of this thesis, analysis of software non-functional qualities is made, possible architectural styles are investigated. Review of existing methods, that allow assessment of qualities of software systems, is performed. In the end of this thesis examples of application of this method are given.

Keywords: software architecture, architectural styles, agent based modelling, simulation, non-functional qualities, distribution over a network, PACELC theorem, CAP theorem.

TURINYS

ĮVADAS

Aktualumas ir naujumas

Plačiausiai naudojamuose programų sistemų kūrimo modeliuose yra numatyta veikla, kurios rezultatas yra programų sistemos architektūra. Priklausomai nuo pasirinkto programų sistemų kūrimo modelio gali skirtis architektūros kūrimo veiklos pobūdis ir eiga. Pavyzdžiui, pasirinkus krioklio ar jam artimą (pvz. iteratyvųjį krioklio) modelį, sistemos architektūra pradedama kurti pasibaigus reikalavimų analizės fazei ir turi būti sukurta iki realizavimo (programavimo) fazės pradžios [Roy70]. Agile modelyje architektūra pradedama kurti „nulinėje“ iteracijoje, tačiau tuo metu nesileidžiama į smulkias detales: apibrėžiamos verslo esybės, pasirenkamos technologijos (pvz. karkasai) ir apibrėžiami jų sąryšiai bei nusakoma, kaip sistema bus diegiama [Ambb]. Smulkesnės architektūros detalės šiame modelyje apibrėžiamos kiekvienos iteracijos pradžioje [Amba]. Ekstremalusis programavimo modelis, kuris yra vienas iš Agile tipų, siekia kiek įmanoma labiau nukelti svarbius sprendimus į ateitį ir palaiko tokią architektūrą, kuri minimaliai tenkina dabartinius poreikius [LRM+07]. Atmetus kraštutinius modelius, tokius kaip ekstremalusis, galima įžvelgti poreikį priimti architektūrinius sprendimus anksčiau, nei tai padaryti priverčia aplinkybės – neatitikimai nefunkciniams reikalavimams.

Vienas iš pradinių bei svarbiausių architektūrinių sprendimų – architektūrinio stiliaus ar kelių stilių parinkimas kuriamai programų sistemai. Architektūrinis stilius – pakartotinai panaudojamas, sprendimų ir apribojimų rinkinys, kuris yra taikomas konkrečiai architektūrai siekiant, kad ji turėtų pageidaujamas savybes [ISR]. Pasirinkti architektūriniai stiliai nulemia ar sukurta programų sistema tenkins nefunkcinius reikalavimus, tokius kaip: sparta, pasiekiamumas (angl. *availability*), išplečiamumas (angl. *scalability*) bei patikimumas [Har11].

Šiuo metu, programų sistemų architektai neretai susiduria su problemomis, kurios kyla bandant užtikrinti patenkinamą spartą ir pasiekiamumą esant dideliam kiekiui naudotojų, kurie naudojasi sistema vienu metu. Gali kilti mintis spręsti šią problemą didinant serverių pajėgumą pvz. naudojant vis galingesnius procesorius, daugiau darbinės atminties ar panašiai. Tačiau dabartinės tendencijos rodo, jog linkstama rinktis kitokį sprendimo būdą – naudojant vis daugiau į bendrą tinklą sujungtų, tačiau ne pačių brangiausių, serverių [GLM+08]. Tai reiškia, jog programų sistemų architektūros turi būti suprojektuotos taip, kad sistemos veikdamos išnaudotų turimus fizinius serverius – sistemos dalys turėtų būti išskirstytos juose.

Viena iš reikalavimų inžinerijos užduočių – surinkti bei analizuoti nefunkcinius reikalavimus. Nefunkciniai reikalavimai gali būti išreiškiami tiek kokybiškai, tiek kiekybiškai [CNY+00]. Kuriant sistemos architektūrą privaloma ją sukurti tokią, kad ji būtų pajėgi patenkinti abiejų tipų iškeltus nefunkcinius reikalavimus. Tai galima padaryti tik turint metodus, kurie leistų architektūras vertinti nefunkcinių reikalavimų atžvilgiu. Tokių metodų paieška ir kūrimas prasidėjo jau seniai – ne vėliau nei prieš 21 metus [KAB+96] ir tai vis dar aktualu iki šiol (viena iš konferencijos ICSA 2017 temų buvo „Architektūrų vertinimas“ [ICSA17]).

Agentais grįstas modeliavimas naudojamas tyrinėti reiškiniams socialinėse, fizinėse bei biologinėse sistemose pvz. oro eismo kontrolės centrų veiklai, epidemijų plitimui modeliuoti. Jį būtų galima priskirti į simuliacijomis grįstų metodų kategoriją. Šaltiniuose teigiama, jog agentais grįstas modelis tinka kai [MN09]:

- Agento priiminėjami sprendimai ir elgsena gali būti apibrėžti;
- Svarbu, jog agentas galėtų savarankiškai prisitaikyti ir keisti savo elgseną;
- Svarbu, jog agentas turėtų dinaminį ryšį su kitais agentais;
- Svarbu, jog būtų galima nesudėtingai plėsti modelį naujais agentais, ryšiais ir būsenomis.

Taip pat, tokiais atvejais, kai [KB12]:

- Neįmanoma nuspėti visos sistemos elgsenos makro lygiu pagal tai, kaip elgiasi pavieniai sistemos elementai;
- Agentų vidinė struktūra, sąveikos galimybės – sudėtingos.

Šie punktai suteikia pagrindą programų sistemų architektūrinius stilius modeliuoti agentais. Taip pat, yra sėkmingai atliktų tyrimų, kuriuose buvo naudojamas agentais grįstas modeliavimas tiriant programų sistemų, kuriose yra komponentų su būsenomis, architektūras [Sil16] bei lyginant įvykiais grįstas architektūras su architektūromis orientuotomis į paslaugas [Mik14].

Pagrįsti pasirinkimą modeliuoti architektūrinius stilius agentais, galima ir palyginant agentinę paradigmą su labiau įprasta ir dažniau naudojama objektine paradigma. Tai yra, atsakyti į klausimą, kodėl verta rinktis mažiau paplitusią bei ne tokią populiarią paradigmą. Privalumai išryškėja lyginant pagrindinius šių paradigmų elementus – agentus ir objektus. Agentai yra autonomiški, kokie gali būti ir architektūriniai elementai: jie patys gali inicijuoti veiksmus, be išorinio poveikio ar įsikišimo arba reaguoti į aplinkoje įvykstančius įvykius. Objektai atlieka veiksmus tik tokiu atveju, kuomet yra iškviečiami jų metodai. Agentai, kaip ir architektūriniai elementai, gali pasižymėti tam tikro laipsnio nenuspėjamumu (arba nedeterminizmu). Pavyzdžiui, agentas gali atsisakyti atlikti jam

skirtą užduotį ir tai nebus laikoma klaida. Objektinės paradigmos atveju – tai būtų klaida. Objektinėje paradigmoje objektas sukuriamas pagal šabloną – klasę ir savo funkcijų negali keisti. Agentinėje paradigmoje agentas gali turėti keletą rolių tuo pačiu ar skirtingais laikais, kaip ir programų sistemos dalis: pavyzdžiui, ji gali atsakinėti į naudotojo užklausas ir tuo pat metu, foniniu režimu, periodiškai apdoroti didelius kiekius duomenų. Taip pat, agentams yra įprasta prisitaikyti prie prieinamų išteklių kiekio: jie gali išnykti iš sistemos, kuomet išteklių trūksta, ir savarankiškai atsirasti vėl, kai (jei) išteklių pakanka [Ode02]. Taip gali elgtis ir programų sistemos dalys.

Sistemų architektūros gali būti analizuojamos dviem būdais: naudojant dinaminius arba statinius metodus. Dauguma statinių metodų bando įrodyti sistemos korektiškumą, kas gali būti naudinga, tačiau reikalauja labai detalaus ir tikslaus sistemos modelio. Dinaminiai, kita vertus, mėgina atspindėti sistemos elgseną veikimo metu, leidžia palyginti aukšto lygio architektūrinius sprendimus bei nereikalauja visiškai tikslaus sistemos modelio, kad būtų išlaikyta modelio nauda ar vertė [EMM07]. Verta paminėti įrankių rinkinį XTEAM, kuris yra dinaminės architektūros analizės realizacija, kuri remiasi simuliacija [EMM07]. Agentais grįstą modeliavimą galima taip pat priskirti dinaminei architektūros analizei. Nors praktikoje tokie architektūrų analizės metodai ir nėra plačiai paplitę, tačiau galima daryti prielaidą, jog taip analizuojant architektūrinius sprendimus (pvz. stilių pasirinkimą) būtų sutaupoma laiko išvengiant sistemos realizacijų (prototipų) ankstyvose projekto stadijose, išvengiama bandymo įvertinti nefunkcines sistemų savybes realiose aplinkose (kas reikalauja papildomų įrankių) bei apsisaugoma nuo nefunkcinių reikalavimų įverčių nuokrypių atsirandančių ne dėl architektūrinio lygmens sprendimų, o dėl pvz. prototipui pasirinktų technologijų.

Darbo tikslas

Sukurti metodą, kuris leistų vertinti architektūrinių stilių, pasižyminčių sistemos dalių išskirstymu tinkle, programų sistemoms suteikiamas nefunkcines veikimo savybes, pasitelkiant agentais grįstą modeliavimą.

Darbo uždaviniai

1. Išnagrinėti programų sistemų bei tinklų savybių vertinimą remiantis modeliavimu agentais ir šio modeliavimo galimybes, ypatybes bei privalumus ir trūkumus lyginant su kitais metodais, apžvelgiant jau esamus bandymus tai atlikti;
2. Išnagrinėti, kokias savybes programų sistemoms, pasižyminčioms komponentų išskirstymu tinkle, suteikia architektūriniai stiliai, jų trūkumus bei privalumus, galimą klasifikaciją;

3. Sukurti metodą, kuriuo vadovaujantis būtų galima sėkmingai vertinti architektūrinių stilių išskirstytoms tinkle programų sistemoms suteikiamas nefunkcines veikimo savybes;
4. Išnagrinėti galimus agentinio modelio validavimo būdus ir pasirinkti tinkamą sukurtam modeliui;
5. Validuoti sukurtąjį metodą;
6. Aprašyti metodo pritaikymo pavyzdį, kurio metu bus įvertintas rinkinys architektūrinių sprendimų pasižyminčių komponentų išskaidymu tinkle, nefunkcinių reikalavimų atžvilgiu.

Laukiami darbo rezultatai

1. Sukurtas metodas įgalinantis vertinti architektūrinių stilių išskirstytoms tinkle programų sistemoms suteikiamas savybes;
2. Atliktas metodo validavimas;
3. Aprašytas metodo pritaikymo pavyzdys, kurio metu įvertintas rinkinys architektūrinių sprendimų pasižyminčių komponentų išskaidymu tinkle, nefunkcinių reikalavimų atžvilgiu.

1. PROGRAMŲ SISTEMŲ BEI TINKLŲ SAVYBĖS

Kadangi šio darbo tikslas yra sukurti metodą, kuris leistų vertinti architektūrinių stilių, pasižyminčių sistemos dalių išskirstymu tinkle, programų sistemoms suteikiamas savybes, būtina išnagrinėti, kokiomis savybėmis apskritai gali pasižymėti pačios programų sistemos bei kompiuteriniai tinklai. Taip pat, reikia nustatyti, kokioms programų sistemų savybėms turi įtakos architektūriniai stiliai ir tinklų charakteristikos.

1.1. Programų sistemų savybės

Naujausias Tarptautinės standartizacijos organizacijos (angl. *ISO*) standartas apibrėžiantis kokybines programinės įrangos charakteristikas – ISO/IEC 25010. Šis standartas apibrėžia du kokybės modelius: naudojimo kokybės ir produkto kokybės. Produkto kokybės modelis susidaro iš šių charakteristikų [ISO11]:

- Funkcinis atitinkamumas (angl. *functional suitability*);
- Veikimo našumas (angl. *performance efficiency*);
- Suderinamumas (angl. *compatibility*);
- Panaudojamumas (angl. *usability*);
- Patikimumas (angl. *reliability*);
- Saugumas (angl. *security*);
- Prižiūrimumas (angl. *maintainability*);
- Perkeliamumas (angl. *portability*).

Kiekviena iš šių charakteristikų turi nuo dviejų iki šešių subcharakteristikų. Naudojimo kokybės modelis apibrėžia šias charakteristikas [ISO11]:

- Efektyvumas (angl. *effectiveness*);
- Našumas (angl. *efficiency*);
- Pasitenkinimas (angl. *satisfaction*);
- Laisvė nuo rizikų (angl. *freedom from risk*);
- Konteksto aprėptis (angl. *context coverage*).

Kiekviena iš naudojimo kokybės modelio charakteristikų turi iki trijų subcharakteristikų.

Visos produkto kokybės charakteristikos, išskyrus funkcinį atitinkamumą, gali tiesiogiai priklausyti nuo pasirinkto architektūrinio stiliaus. Tai pagrindžia keletas tyrimų, kuriuose buvo

siejamos programų sistemų savybės su architektūriniais stiliais [WY12] [ERR11]. Naudojimo kokybės modelio apibrėžiamos savybės nėra tiesiogiai susijusios su architektūriniais stiliais, nes jos yra išvestinės ir orientuotos į naudotojų pojūčius, pvz.: patiriamas malonumas, pasitikėjimas sistema, sunaudoti ištekliai naudojantis sistema ir t.t.

Programų sistemų savybės gali būti kategorizuojamos [MHH+09]:

- Sistemos savybės (angl. *system qualities*);
- Naudotojo savybės (angl. *user qualities*);
- Veikimo savybės (angl. *run-time qualities*);
- Projektavimo savybės (angl. *design qualities*).

Kadangi šio darbo tikslas yra sukurti metodą, kuri leistų vertinti architektūrinių stilių išskirstytoms programų sistemoms suteikiamas nefunkcines veikimo savybes, taigi galimas tyrimo objektas yra: veikimo našumas, patikimumas, saugumas ir suderinamumas.

Tam, kad būtų aiškus galimas tyrimo objektas, būtina detaliau panagrinėti kiekvieną iš minėtų veikimo savybių – detalizuoti jų subcharakteristikas. Veikimo našumo subcharakteristikos [ISO11]:

- Našumas laiko atžvilgiu (angl. *time-behavior*). Nurodo, kiek sistema tenkina spartos (laiko atžvilgiu) reikalavimus;
- Našumas išteklių atžvilgiu (angl. *resource utilization*). Nurodo, kiek sistema tenkina išteklių sunaudojimo reikalavimus;
- Pajėgumas (angl. *capacity*). Nurodo, kiek sistema patenkina jai iškeltus maksimalaus pajėgumo reikalavimus.

Patikimumo subcharakteristikos:

- Branda (angl. *maturity*). Nusako, kiek sistema tenkina patikimumo poreikius esant normaliam veikimui, apkrovai;
- Pasiekiamumas (angl. *availability*). Nusako, kiek sistema yra pasiekiama, kai naudotojams to reikia;
- Atsparumas klaidoms (angl. *fault tolerance*). Nusako, kiek sistema gali funkcionuoti nepaisant programinės ar techninės įrangos klaidų;

- Atsistatomumas (angl. *recoverability*). Nusako, kiek sistema gali atstatyti duomenis ir savo būseną po veiklos sutrikimo.

Saugumo subcharakteristikos:

- Konfidencialumas (angl. *confidentiality*). Vertina, kiek sistema sugeba duomenis apsaugoti nuo neautorizuotų naudotojų prieigos;
- Vientisumas (angl. *integrity*). Vertina, kiek sistema geba apsaugoti nuo neteisėto duomenų pakeitimo;
- Neatšaukiamumas, nepaneigiamumas (angl. *non-repudiation*). Vertina, kiek atliktų veiksmų ar įvykusių įvykių nebegalima atšaukti;
- Atskaitomybė (angl. *accountability*). Vertina, kiek sistema gali užtikrinti atsekamumą tarp subjekto ir jo veiksmų;
- Autentiškumas (angl. *authenticity*). Vertina, kiek sistema gali užtikrinti išteklių ar subjektų autentiškumą. Tai yra, kad subjektas yra tas, kuo teigia esantis.

Suderinamumo charakteristikos:

- Koegzistavimas (angl. *co-existence*). Nurodo, kiek sistema gali atlikti savo užduotis, kuomet ji dalinasi ištekliais ar bendra aplinka su kitomis sistemomis, neigiamai nepaveikdama jų;
- Sąveika (angl. *interopability*). Nurodo, kiek dvi ar daugiau sistemų gali tarpusavyje keistis informacija ir ją naudoti.

1.2. Tinklų savybės

Tyrinėjant programų sistemų architektūras, kurios išskaido savo komponentus skirtinguose fiziniuose serveriuose, ypatingai yra aktualios tinklo, kuris jungia tuos komponentus, savybės. Tai įrodo darbai, kurie konstruoja karkasus, leidžiančius atskirai modeliuoti tinklus ir programų sistemų architektūras, o po to simuliuoti visos sistemos veikimo savybes [VDT+07]. Kaip ir su programų sistemų savybėmis, šiame darbe aktualios tik tinklų veikimo savybes. Yra penkios tinklų charakteristikų grupės [McC07]:

- Pajėgumas (angl. *capacity*); Nurodo gebėjimą perduoti informaciją. Susijusios charakteristikos: bendrasis pralaidumas (angl. *bandwidth*) bei pralaidumas (angl. *throughput*). Pirmoji charakteristika nurodo praeinantį bendrąjį duomenų kiekį per laiko vienetą, o antroji – praeinantį naudingų duomenų kiekį per laiko vienetą.

- Uždelsimas (angl. *delay*); Nurodo laiką, kurio reikia duomenų vieneto (bitas, baitas, paketas ir pan.) perdavimui. Uždelsimas gali būti kelių tipų, pagal šaltinį, iš kurio jis kyla:
    - Ištransliavimo (angl. *transmission*). Tai uždelsimas, kylantis tuomet, kai duomenys yra išsiunčiami („patalpinami į laidą“);
    - Perdavimo (angl. *propagation*). Tai uždelsimas, kuris kyla, kuomet duomenys yra perduodami tinklu („keliauja laidu“);
    - Eilės (angl. *queuing*). Uždelsimas, kuris kyla, kuomet duomenys laukia, kol bus išsiųsti;
    - Apdorojimo (angl. *processing*). Uždelsimas, kylantis kai yra apdorojamos paketų antraštės.

  Visi šie uždelsimo tipai prisideda prie gaišties laiko (angl. *latency*) augimo. Taip pat, daug įtakos turi ir kita uždelsimo savybė – drebėjimas (angl. *jitter*). Drebėjimas nurodo uždelsimo pokyčius laike.
- Patikimumas (angl. *reliability*); Statistinis indikatorius, kuris nurodo nenumatytų tinklo sutrikimų (kurie visiškai sutrikdo jo veikimą) dažnį.
- Prižiūrimumas (angl. *maintainability*); Statistinis indikatorius, kuris nurodo, kiek vidutiniškai laiko reikia, kad tinklas būtų atstatytas po jo sutrikimo.
- Pasiekiamumas (angl. *availability*). Tai indikatorius, kuris išvedamas iš patikimumo ir prižiūrimumo. Vidutinis laikas tarp sutrikimų dalinamas iš vidutinio taisymo ir laiko tarp sutrikimų sumos.

Norint pagrįsti gaišties laiko bei pralaidumo įtaką išskirstytoms programų sistemoms, galima pasiremti tyrimu, tiriančiu gaišties laiko įtaką išskirstytoms duomenų bazėms [Joh00]. Jame iškeliama hipotezė, jog esant dideliam pralaidumui, riboja gaišties laikas, o ne pats pralaidumas. Tyrimas hipotezę patvirtina. Akivaizdu, kad kitos tinklo charakteristikos, tokios kaip patikimumas, turi didžiulę įtaką išskirstytoms programų sistemoms, nes nutrūkus dalies ar viso tinklo veikimui programų sistemų komponentai nebegali pasiekti vienas kito.

1.3. Išskirstytų tinkle programų sistemų veikimo savybės

Pagal ISO 20510 produkto kokybės modelį sudarančias charakteristikas galima vertinti visas programų sistemas. Tačiau šiame darbe yra aktualios būtent išskirstytų tinkle programų sistemų veikimo savybės. Išskirstytos tinkle programų sistemos pasižymi tuo, kad skirtingai nuo viename

fiziniame serveryje veikiančių programų sistemų, ryšiai tarp komponentų gali nutrūkti. Tai gali įvykti sutrikus tinklo veikimui, o to priežastys Gill et al. straipsnyje skirstomos į [GJN11]:

- Jungties klaidas (angl. *link failure*). Kuomet susijungimai tarp dviejų skirtingų įrenginių nutrūksta. Tai gali įvykti tiek dėl programinės, tiek dėl techninės įrangos gedimų, konfigūracijos klaidų, o pagrinde dėl nepavykusių tinklo susijungimų (taip gali atsitikti dėl pvz. problemų su laidais);
- Įrenginio klaidas (angl. *device failure*). Kai įrenginys, turintis nustatyti maršrutą ar persiųsti srautus, nustoja tai daręs. Įrenginiai gali būti: apkrovos paskirstytojai (angl. *load balancer*), maršrutizatoriai (angl. *router*), tinklo jungikliai (angl. *network switch*) ir kt. Jų veikla gali nutrūkti juos išjungus planuotam aptarnavimui ar sugedus techninei įrangai.

Tinklo veikimo problemų praktikoje neįmanoma išvengti, galima tik analizuoti problemų priežastis (tą ir daro Gill et al. savo straipsnyje) bei bandyti jas kiek įmanoma eliminuoti.

Kadangi ryšiai tarp komponentų gali nutrūkti programų sistemos veikimo metu, sistemos architektams būtina žinoti, kokios sistemos savybės tokiu atveju pakinta (t.y. turėti galimybę prognozuoti savybių pasikeitimus). Gilbert ir Lynch straipsnyje [GL02], parašytame remiantis Eric Brewer kviestiniame pranešime išsakyta idėja (vėliau ji buvo pavadinta CAP teorema), jog programų sistema gali pasižymėti dvejomis savybėmis iš trijų: vientisumu (angl. *consistency*), pasiekiamumu (angl. *availability*) ir atsparumu ryšio nutrūkimams tarp komponentų (angl. *partition-tolerance*), teigia, jog šios savybės yra vienos svarbiausių išskirstytoms tinklo aplikacijoms, tinklo paslaugoms (angl. *web services*). Būtina paminėti, kad šios savybės yra veikimo savybės, nes pasireiškia tik programų sistemos veikimo metu. Kadangi trys minėtosios sąvokos neretai turi skirtingą prasmę skirtinguose kontekstuose, privalu pateikti tokius apibrėžimus, kurie naudojami CAP teoremoje [GL02]:

- Vientisumas – jei ši savybė yra užtikrinama, tuomet visi atlikti veiksmai programų sistemoje turi vieną bendrą eilę, kuri nurodo, kuris po kurio veiksmo buvo atliktas, tarsi sistema veiktų viename mazge (angl. *node*), o ne keliuose lygiagrečiai;
- Pasiekiamumas – kiekviena užklausa, kuri pasiekia „gyvą“ mazgą programų sistemoje, turi gauti atsakymą (tačiau nenurodoma, per kokį laiko tarpą);
- Atsparumas ryšio trūkiams tarp komponentų – savybė leidžianti prarasti bet kokį kiekį pranešimų siųstų iš vieno mazgo į kitą.

Kadangi tinklo veikimo sutrikimai yra neišvengiami, vadinasi, atsisakyti atsparumo ryšio trūkiams – negalima. Todėl iš esmės teorema teigia, jog esant ryšio trūkiams tarp komponentų, reikia rinktis, kokią savybę išpildyti labiau – vientisumą ar pasiekiamumą. Šią išvadą patvirtina ir Abadi straipsnis [Aba12], kuriame jis teigia tą patį.

Abadi pasiūlo ir CAP teoremos modifikaciją – PACELC. Ji aiškinama dvejais sąlygos sakiniais:

- Jei nutrūko ryšys tarp komponentų, kokią savybę tuomet sistema renkasi užtikrinti – pasiekiamumą ar vientisumą?
- Jei ryšys nėra nutrūkęs, sistema renkasi užtikrinti vientisumą ar greitą atsako laiką?

Antrasis sąlygos sakinys galioja tik tuomet, jei programų sistema turi galimybę dubliuoti duomenis. Dubliuojant duomenis užtikrinamas greitas atsako laikas (nes pasiskirsto tenkanti apkrova), tačiau prarandamas vientisumas, nes vienu metu atnaujinti visas duomenų dublikatus – neįmanoma. Visuomet bus laiko tarpas (galbūt nykstamai trumpas), kurio metu bus atnaujinami dublikatai, o to pasekmėje, juose esantys duomenys nebus vienodi. Šis reiškinys vadinamas galutiniu vientisumu (angl. *eventual consistency*) ir gana detaliai nagrinėjamas knygoje [Bur14].

Abadi straipsnyje kalba apie išskirstytas duomenų bazes (angl. *distributed database system*), tačiau turint omenyje tai, jog jo pasiūlyta PACELC yra CAP teoremos modifikacija (patikslinimas), o Brewer savo pranešime [Bre00] CAP teoremą taikė bet kurioms programų sistemoms, kurių komponentai turi bendros informacijos (angl. *shared-data system*), galima daryti išvadą, jog ir PACELC tinka tokio pat pobūdžio sistemoms. Kitaip tariant, PACELC sąlygos sakiniai tinka apibūdinti programų sistemoms, kurių komponentai turi bendras būsenas.

1.4. Programų sistemų komponentų būsenos

Siliūno magistriniame darbe [Sil16] yra apibrėžiama, kas tai yra būsenos bei kaip jos yra kategorizuojamos. Būsena – informacija atsiradusi dėl sistemos veiklos, kuri yra saugoma sistemoje bei daro įtaką tolimesnei sistemos veiklai. Būsena aktuali nes gali turėti įtakos sistemos veikimui, jos kokybinėms charakteristikoms. Tiek šiame, tiek Siliūno darbe yra aktuali tik tokia būsena, kurią galima nustatyti ir nuskaityti sistemos veikimo metu. Kategorizuojama gali būti pagal apimtį ir laiką [Sil16]:

- Apimties dimensijos kategorijos:

- Komponentą apimanti būsena. Tai informacija, kuri saugoma komponente ir naudojama koreguoti komponento atsaką. Ją keisti gali tik pats komponentas. Kategorizuojama:
  - Protokolo būsena. Informacija apie galimybę priimti tam tikro pobūdžio užklausas;
  - Vidinė būsena. Komponento viduje nustatyta informacija. Ji yra matoma iš išorės, bet ją keisti gali tik pats komponentas. Naudojama komponento veiklai koreguoti;
  - Priskyrimo būsena. Informacija nustatoma apie aplinką (infrastruktūrą) išdėstymo metu;
  - Konfigūracija. Komponento atributų reikšmes nusakanti informacija, nustatoma inicijavimo metu.
- Sistemą apimanti būsena. Visiems komponentams prieinama informacija, kuri naudojama sistemos veiklos koordinacijai. Kategorizuojama:
  - Globali būsena. Informacija prieinama visiems komponentams vykdymo metu;
  - Priskyrimo būsena. Informacija prieinama visiems komponentams vykdymo metu, sukuriama išdėstymo metu;
  - Konfigūracija. Sistemos paleidimo metu nustatoma informacija.
- Vartotoją apimanti būsena. Informacija susieta atskirai su kiekvienu naudotoju. Leidžia keisti sistemos veikimą priklausomai nuo to, koks naudotojas į ją kreipėsi. Kategorizuojama:
  - Sesijos būsena. Naudotojo informacija, kuri egzistuoja kartu su sesija. Sesijai išnykus – išnyksta ir informacija;
  - Nuolatinė būsena. Naudotojo informacija, kuri egzistuoja visą sistemos gyvavimo laikotarpį.

- Laiko dimensijos kategorijos [Sil16][HBR14]:
  - Inicijavimo. Laiko tarpas, kuomet pats komponentas yra sukuriamas ar pvz. sistema surenkama iš komponentų;
  - Išdėstymo (diegimo);
  - Vykdymo. Laiko tarpas, kuomet sistema vykdo savo veiklą.

Pateiktoje būsenų klasifikacijoje nėra kategorijos, kuri apibūdintų būsenas, kurios yra bendros tarp dviejų ar daugiau, bet nebūtinai tarp visų komponentų, kaip kad visą sistemą apimančios

būsenos. Taip pat, tokios kategorijos būsenas turėtų būti leidžiama valdyti (skaityti, atnaujinti) visiems komponentams, tarp kurių ta būsena yra bendra. Ši kategorija aktuali šiam darbui, nes tokios būsenos aktualios išskirstytoms sistemoms, o jų architektūra neretai projektuojama taip, kad duomenys būtų taip pat išskirstyti, siekiant patenkinti tam tikras kokybines charakteristikas, tokias kaip aukštas pasiekiamumas. Tą patį teigia ir Scott et al. straipsnyje [SCD+03], kuriame bando pagrįsti, jog sistemos su bendromis būsenomis įgis vis didesnį populiarumą, kas didins mechanizmų, leidžiančių automatizuoti bendrų būsenų valdymą programų sistemose, paklausą.

Bendrų būsenų valdymą (skaitymą, atnaujinimą) išskirstytose sistemose nagrinėja Burckhardt knygoje [Bur14]. Pagrindinė problema bendrų būsenų valdyme yra tai, jog atnaujinimo operacija nėra komutatyvi. Tai yra, jei turime du konkurentinius atnaujinimus tai pačiai būsenai, svarbi yra tvarka, kuria atnaujinimai bus įvykdyti. Priklausomai nuo duomenų tipo, galimi įvairūs šios problemos sprendimo būdai. Pavyzdžiui, dublikuotam registrui (šiuo atveju, tai duomenų tipas, kuris saugo vieną reikšmę, kurią leidžią įrašyti ir nuskaityti), Burckhardt siūlo du sprendimus:

- Paskutinis atnaujinimas laimi (angl. *Last-Writer-Wins*);
- Kelių reikšmių saugojimas. Saugomos kelios reikšmės ir iš jų leidžiama pasirinkti naudotojui, kurią laikyti „paskutine“.

Dublikuotoms aibėms kyla problema, kaip elgtis situacijoje, kuomet tas pats elementas yra konkurentiškai trinamas ir įrašomas. Vienas iš variantų, esant tokiai situacijai – leisti įrašymo operacijai laimėti (angl. *Add-Wins set*), nes kitu atveju, laimėjus paskutiniam atnaujinimui, gali būti prarasti duomenys. Yra duomenų tipų, kuriems tokio pobūdžio problemų kilti negali, pvz. skaitiklis (duomenų tipas, kurio pagrindinė operacija yra „padidinti“ (angl. *increment*)), nes taikoma operacija „padidinti“ yra komutatyvi.

2. PROGRAMŲ SISTEMŲ ARCHITEKTŪRA

2.1. Architektūros apibrėžimų analizė

Šaltiniuose galima rasti labai įvairių apibrėžimų, kurių tikslas – nusakyti, kas yra programų sistemų architektūra. Sudėtingiausia apibrėžiant architektūrą yra nubrėžti ribą, kur baigiasi architektūra ir prasideda žemesnio lygio sprendimai. Šiame skyriuje apžvelgsime keletą architektūros apibrėžimų.

Programinės įrangos inžinerijos institutas (angl. *Software Engineering Institute, Carnegie Mellon University*) yra išskirstęs programinės įrangos architektūros apibrėžimus į tris grupes pagal kilmę [SEI10]: modernius, klasikinius (rastus vertinguose, žinomuose šaltiniuose) ir bibliografinius. Moderniuose šaltiniuose rasti:

- Bass, Clements ir Kazman knygoje [BCK03] architektūra apibrėžiama taip: sistemos struktūra ar struktūros, apimančios programinės įrangos elementus, išoriškai matomas tų elementų savybes ir ryšius tarp tų elementų. Knygoje pateikiamas apibrėžimo paaiškinimas. Išoriškai matomos savybės – elementų teikiamos paslaugos, veikimo charakteristikos, klaidų apdorojimas, išteklių naudojimas ir t.t. Architektūra nusako tai, kaip elementai vienas su kitu susiję, kaip bendrauja tarpusavyje, bet praleidžia tą informaciją, kuri nesusijusi su sąveika tarp elementų. Taip pat, kiekviena sistema turi architektūrą, net jei ji nėra žinoma.
- Standarte IEEE (angl. *Institute of Electrical and Electronics Engineers*) 1471 yra pateikiamas toks apibrėžimas: fundamentali sistemos struktūra, išreikšta sistemos komponentais, jų tarpusavio ryšiais ir aplinka bei principais, kurie apibrėžia sistemos evoliuciją ir dizainą.

Klasikiniuose šaltiniuose rasti:

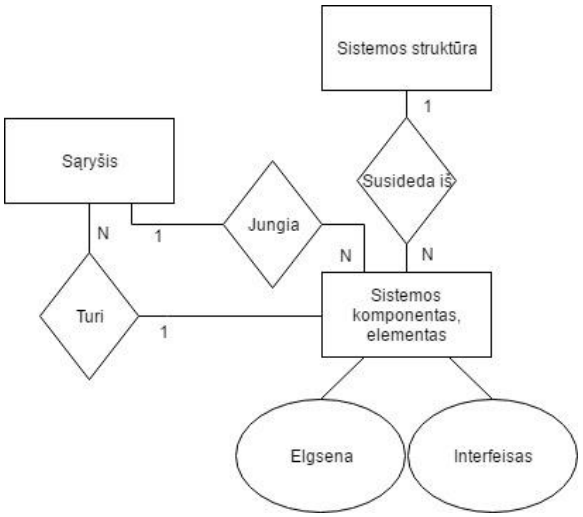
- Kruchten knygoje [Kru99] pateikiamas toks apibrėžimas: architektūra yra aibė reikšmingų sprendimų programinės įrangos sistemos struktūroje, struktūrinių elementų iš kurių susideda sistema ir jų interfeisų pasirinkimas, struktūrinių elementų elgsenos (kiek tai susiję su bendradarbiavimu tarp elementų) nustatymas, struktūrinių elementų kompozicija į stambesnes sistemas ir architektūrinio stiliaus pasirinkimas, kuris rodo kryptį atliekant prieš tai minėtus veiksmus.

- Hayes-Roth vienoje iš ataskaitų [HR94] teigia, jog architektūra, tai abstrakti sistemos specifikacija, susidedanti iš funkcinių komponentų, kurių elgsena aprašyta bei interfeisų ir komponentų tarpusavio sąryšių.

Analizuojant visus čia pateiktus architektūros apibrėžimus išryškėja keletas raktinių sąvokų arba kitaip, esybių ir jų atributų, kurios egzistuoja visuose apibrėžimuose:

- Sistemos struktūra;
- Sistemos komponentas, elementas;
- Komponento, elemento interfeisas;
- Komponento, elemento elgsena;
- Komponentų, elementų tarpusavio sąryšis.

Architektūros apibrėžimą galima atvaizduoti esybių-ryšių diagrama:



1 Pav.: Architektūros apibrėžimo esybių-ryšių diagrama. Naudojama Chen notacija.

Rozanski ir Woods knygoje [RW09] pateikiamas būdas, kaip architektūra gali būti struktūrizuotai apibrėžiama, laikantis interesų atskyrimo principo (angl. *separation of concerns*). Tam naudojami architektūriniai požiūriai (angl. *viewpoint*), kurie leidžia architektūrą apibrėžti pagal šiuos interesus [RW09]:

- Funkcinis (angl. *functional*). Apibrėžia sistemos funkcinius elementus, jų atsakomybes, interfeisus;
- Informacinis (angl. *information*). Apibrėžia, kaip sistema saugo, valdo ir paskirsto informaciją;

- Konkurentiškumo (angl. *concurrency*). Apibrėžia, kurios sistemos dalys gali veikti lygiagrečiai, kaip lygiagretumas yra valdomas;
- Kūrimo (angl. *development*). Apibrėžia programų sistemų kūrimo procesui aktualią architektūrą;
- Diegimo (angl. *deployment*). Apibrėžia aplinką, į kurią sistema bus diegiama;
- Operacinis (angl. *operational*). Apibrėžia, kaip sistema bus administruojama, palaikoma.

2.2. Architektūrinio stiliaus apibrėžimų analizė

Aukščiau pateiktame Kruchten apibrėžime yra minimas architektūrinis stilius kaip kryptis, kurios reikia laikytis kuriant architektūrą, tačiau toks apibrėžimas per daug abstraktus. Randama gerokai mažiau šaltinių, kuriuose architektūrinis stilius būtų apibrėžtas, lyginant su programų sistemos architektūros apibrėžimu. Vieni pirmųjų, kurie pasiūlė šią sąvoką yra Garlan ir Shaw savo knygoje [GS94]: architektūrinis stilius nustato komponentų ir jungčių „žodyną“, kuriuo galima naudotis kuriant architektūrą pagal tą stilių, laikantis stiliuje numatytų apribojimų, kaip tuos komponentus ir jungtis galima derinti tarpusavyje. Kitas apibrėžimas, rastas Fielding disertacijoje [Fie00]: suderinta aibė architektūrinių apribojimų, kurie suvaržo architektūrinių elementų vaidmenis/ypatybes bei apibrėžia leistinus ryšius tarp elementų toje architektūroje, kuri yra kuriama pagal tą stilių. Iš esmės, galime pastebėti, jog abu apibrėžimai semantiškai tapatūs.

2.3. Architektūriniai stiliai, jų klasifikacija

Architektūriniai stiliai gali būti skirstomi į kategorijas pagal tai, į kokį aspektą jie yra orientuoti [MHH+09]:

- Komunikacijos (angl. *communication*). Pavyzdžiai: žinučių jungties (angl. *message bus*), įvykiais grįsta architektūra (angl. *event-driven*). Nurodo taisykles, kaip sistemos komponentai komunikuoja tarpusavyje. Kompanijos Oracle straipsnyje [Sch11] komunikacijos stiliai skirstomi į tris tipus:
    - Laiku grįsti. Sąveika tarp elementų vyksta numatytu laiku;
    - Užklausomis grįsti. Klientas užklausęs laukia, kol serveris pateiks atsakymą ar bent praneš, kad gavo užklausą;
    - Įvykiais grįsti. Apibrėžimas pateikiamas šio darbo sekančiuose skyriuose.
- Diegimo (angl. *deployment*). Pavyzdžiai: kliento/serverio, trijų pakopų (angl. *3-tier*). Nurodo, kaip ir kur sistema yra diegiama;

- Srities (angl. *domain*). Pavyzdžiai: sritimi grįstas projektavimas (angl. *Domain Driven Design*);
- Struktūros (angl. *structure*). Pavyzdžiai: komponentinė, objektinė, sluoksninė (angl. *layered*). Nurodo, iš kokių dalių yra sudaroma architektūra.

Kadangi yra ne viena stilių kategorija (kiekviena iš jų turi savo paskirtį), tai projektuojant architektūras dažnu atveju yra panaudojamas ne vienas architektūrinis stilius, o jų kombinacija, nes reikia nuspręsti ir iš kokių dalių bus sudaryta architektūra, ir kur ji bus diegiama, ir t.t. Tai patvirtina ir Garlan savo knygoje [GS94].

Šiame darbe, remiantis agentais grįsto modelio simuliacija, bus nagrinėjami komunikacijos tipo architektūriniai stiliai, nes simuliacijos metu agentams komunikuojant tarpusavyje kyla įvairių reiškinių makro lygyje.

2.4. Išskirstytų tinkle programų sistemų architektūriniai stiliai

Tiek įprastinių programų sistemų, kurios veikia viename fiziniame kompiuteryje, architektūroms, tiek tinkle išskirstytų programų sistemų architektūroms, komunikacijos architektūriniai stiliai yra aktualūs. Taip yra todėl, kad komunikacija tarp architektūros komponentų visuomet vyksta, nepaisant to, ar tai yra komunikacija pagrindinėje atmintyje ar komunikacija tinklu, skiriasi tik komunikacijos savybės. Dėl skirtingų komunikacijos savybių gali skirtis ir tokių stilių įgyvendinimas: atsirasti papildomų architektūrinių komponentų būtinų stiliaus įgyvendinimui ar pan.

Komunikavimo stiliai gali būti skirstomi pagal sinchronizaciją:

- Sinchroninis komunikavimas. Kuomet siuntėjui išsiuntus užklausą, gavėjas privalo atsakyti iškart neinicijuodamas naujo susijungimo, o siuntėjas aktyviai (nevykdydamas kitos veiklos) laukia atsakymo;
- Asinchroninis komunikavimas. Kuomet siuntėjui išsiuntus užklausą nereikia aktyviai laukti (gali būti vykdoma kita veikla), kol gavėjas atsakys.

Tokio pobūdžio skirstymas turi prasmę, nes pasirinkimas turi didelę įtaką programų sistemų savybėms [Sho16]. Pvz. sinchroninis komunikavimas yra imlesnis laikui, nes asinchroninis leidžia atlikti kelias užklausas vienu metu, todėl galimai sutaupo laiko.

2.4.1. Įvykiais grįstas stilius

Vienas iš galimų asinchroninio komunikavimo stilių yra įvykiais grįstas (angl. *event-driven*). Michelson straipsnyje [Mic06] autorė apibrėžia, kas yra įvykiais grįsta architektūra (angl. *event-driven architecture*):

- Įvykis – tai svarbus dalykas, kuris atsitinka sistemos viduje ar išorėje. Jis gali pranešti apie problemą, artėjančią problemą, galimybę, nukrypimą ar pan.;
- Įvykiais grįstoje architektūroje įvykis pasklinda visiems, o jo gavėjai sprendžia ar ignoruoti, ar atlikti kažkokius tolimesnius veiksmus: iškviesti servisą, paleisti veiklos procesą ar pan.;
- Didžiausias įvykiais grįstos architektūros privalumas – žema komponentų sankiba (angl. *coupling*).

Skirtingose aplinkose įvykiais grįsta architektūra įgyvendinama skirtingai. Kai įvykiais grįsta komunikacija realizuojama objektinėje paradigmoje, įprastu atveju naudojamas stebėtojo šablonas (angl. *observer pattern*). Pavyzdžiui, programavimo kalba Java, nuo pat pirmosios jos versijos, turi pagalbinį interfeisą *Observer* ir klasę *Observable,* kurios leidžia įgyvendinti stebėtojo šabloną. Išskirstytose programų sistemose, įgyvendinant įvykiais grįstą stilių, naudojamas publikavimo-prenumeravimo šablonas (angl. *publish-subscribe pattern*). Eugster et al. straipsnyje [EFG+03] apžvelgia ne vieną šio šablono alternatyvą ir variaciją. Autoriai teigia, jog yra gana sudėtinga lyginti įvairius sprendimus, nes jie būna skirtingo abstrakcijos lygio. Alternatyvos:

- Žinučių perdavimas (angl. *message passing*). Komunikuojama siunčiant ir gaunant žinutes. Retai naudojama kaip pagrindinė priemonė. Tiek siuntėjas, tiek gavėjas turi būti „gyvi“ duomenų perdavimo metu bei siuntėjas turi žinoti, kaip pasiekti gavėją.
- Nuotolinis procedūros kvietimas (angl. *Remote Procedure Call (RPC)*). Sprendimas leidžiantis kreiptis į nutolusią procedūrą taip, lyg tai būtų lokalus kreipimasis. Taip pat, tiek siuntėjas, tiek gavėjas turi būti „gyvi“ perdavimo metu. Siuntėjas turi žinoti, kaip pasiekti gavėją.
- Pranešimai (angl. *notifications*). Sinchroninis kreipimasis skaidomas į dvi dalis: asinchroninį kvietimą ir asinchroninį atsakymą. Kvietimo metu siunčiami parametrai ir nuoroda, kur grąžinti atsakymą. Atsakymas grąžinamas ten, kur buvo nurodyta kvietime. Siuntėjas ir gavėjas turi būti „gyvi“ perdavimo metu. Siuntėjas turi žinoti, kaip pasiekti gavėją.

- Išskirstyta bendra atmintis (angl. *distributed shared memory*). Komunikavimas vyksta per kortežų erdvę (angl. *tuple space*): į ją įdedant ir ištrinant kortežus. Siuntėjas gali nežinoti kur yra gavėjas ir neprivaloma abiem būti „gyviems“ vienu metu.
- Žinučių eilės (angl. *message queuing*). Panašiausia į publikavimo-prenumeravimo šabloną. Skirtumas tik toks, kad siuntėjo siunčiama žinutė gali būti gauta tik vieno gavėjo. Siuntėjas gali nežinoti kur yra gavėjas ir neprivaloma abiem būti „gyviems“ vienu metu. Neužtikrina visiško asinchroniškumo, nes gavėjai gauna žinutes sinchroniškai.

Straipsnyje pateikiamos alternatyvų palyginimo išvados, jog visos šios alternatyvos iš esmės skiriasi ir publikavimo-prenumeravimo šablonas siūlo laisviausią sankibą visais aspektais: laiko, sinchronizacijos bei erdvės.

Publikavimo-prenumeravimo šablono variacijos:

- Tema grįstas (angl. *topic-based*). Jei elementai X ir Y prenumeruoja temą T, vadinasi, visi įvykiai pranešti su antraštėje nurodyta tema T pasieks elementus X ir Y. T dažnu atveju – eilutės (angl. *string*) tipo kintamasis;
- Turiniu grįstas (angl. *content-based*). Prenumeruojama pagal įvykio turinyje esančias pavadinimo-reikšmės poras – joms yra rašomi predikatai;
- Tipu grįstas (angl. *type-based*). Panašu į tema grįstą variaciją, tačiau T – tipas (pvz. objektinės paradigmos klasė).

Variacijų palyginimo išvadose teigiama, jog tema grįsta šablono variacija yra pati primityviausia, tačiau gali būti realizuota efektyviausiai. Turiniu grįsta variacija atvirkščiai – pati išraiškingiausia, tačiau ją sudėtinga įgyvendinti efektyviai.

Norint įgyvendinti publikavimo-prenumeravimo šabloną, reikia pasirinkti tarpinę programinę įrangą (angl. *middleware*). Yra bent dvi specifikacijos, kurių realizacijas būtų galima laikyti tinkama tarpine programine įranga: Išplėstinis žinučių eilės protokolas (angl. *Advanced Message Queuing Protocol (AMQP)*) bei Duomenų paskirstymo tarnyba (angl. *Data Distribution Service (DDS)*). *AMQP* yra gerokai populiaresnis, dažniau naudojamas, protokolas ir turi žymiai daugiau produktų paremtų juo, nei *DDS: RabbitMQ, OpenAMQ, StormMQ, ActiveMQ* ir kiti. *DDS* realizacijos: *OpenDDS, Vortex OpenSplice.*

2.4.2. Užklausomis grįstas stilius

Tai paprastas, galimai sinchroninis (įmanomas ir asinchroninis variantas, tačiau šiame darbe aktualus sinchroninis, kaip alternatyva asinchroniniam, įvykiais grįstam stiliui) komunikavimo tarp skirtingų architektūrinių elementų stilius. Nepaisant to, šį stilių galima įgyvendinti ne vienu būdu, o bent dviem, jau minėtais aukščiau: žinučių perdavimo ir nuotolinio procedūros kvietimo.

Nuotolinis procedūros kvietimas (NPK) gali būti realizuotas įvairiais būdais: po juo gali slėptis įvairūs informacijos perdavimo protokolai (praktikoje, dažnu atveju, tai būna TCP pagrindu veikiantis HTTP, bet gali būti ir tiesiog TCP ar UDP), duomenų formato protokolai (JSON, XML ir kt.). Nuo pasirinktų protokolų priklauso perdavimo patikimumas (pvz. UDP nėra patikimumą užtikrinantis protokolas), sparta (priklausomai nuo pasirinkto duomenų formato, bendras perduodamų duomenų kiekis gali keistis). Protokolų šiai įdėjai įgyvendinti yra labai daug ir įvairių: *CORBA, JSON-RPC, SOAP.* Realizacijų taip pat netrūksta: *Java RMI, Netflix Ribbon* ir kt.

Kitas būdas siųsti užklausas – žinučių perdavimo būdu. Nuo nuotolinio procedūros kvietimo, šis metodas skiriasi tuo, kad dažnu atveju interfeisas turi tik dvi funkcijas - *send* ir *receive*, o tai reiškia, kad nebandoma sudaryti įspūdžio, jog kvietimas yra lokalus. Įgyvendinama gali būti remiantis įvairiais tinklo protokolais, kaip ir NPK atveju. Perduodamų duomenų struktūra, taip pat gali būti įvairi.

Apibendrinant, abu būdai yra labai panašūs, nes jų veikimo savybės priklauso nuo pasirinktų realizacijų: tinklo, duomenų protokolų. Pagrindiniai skirtumai – programos, naudojančios vieną ar kitą būdą, išeities kodo tekste, o tai nėra šio darbo tyrimo objektas.

3. PROGRAMŲ SISTEMŲ SAVYBIŲ VERTINIMO METODAI

3.1. Metodų klasifikacija

Metodai, skirti programų sistemų savybių vertinimui, gali būti kategorizuojami į grįstus [Mår06]:

- Patirtimi. Remiamasi turima kūrėjų ir konsultantų patirtimi. Pastarieji anksčiau susidūrę su panašiomis situacijomis, gali pasakyti, ar architektūra tenkins reikalavimus.
- Simuliacijomis. Remiamasi aukšto lygio, dalies ar visų architektūros komponentų bei aplinkos, realizacija. Tokiu metodu galima nustatyti architektūros veikimo savybes;
- Matematiniais modeliais. Remiamasi matematiniais metodais ir įrodymais, nustatinėjant komponentų veikimo savybes, patikimumą. Gali būti kombinuojama su simuliacija, taip padidinant tikslumą;
- Scenarijais. Vertinama konkreti savybė pagal scenarijų, kuris labai tiksliai aprašo reikalavimą.

3.2. Programų sistemų architektūrų savybių vertinimo metodų palyginimas

Siekiant pasirinkti metodą ar metodiką, kuris leistų tinkamai vertinti architektūrinių stilių programų sistemoms suteikiamas nefunkcines veikimo savybes, reikia išsikelti kriterijus, kuriais būtų galima vertinti metodus. Pasirinkti kriterijai:

- Metodo įeigos pobūdis. Kadangi darbo tikslas yra vertinti stilių suteikiamas savybes tinkle išskirstytoms programų sistemoms, tai metodo įeiga turėtų būti tiesiogiai susijusi su stilių ypatybėmis:
    - Tarpusavio komunikacijos (jungčių) tarp komponentų pobūdžiu ir parametrais;
    - Tinklo konfigūracija, jo parametrais;
    - Komponentų elgsena bei būsenomis.
- Metodo išeigos pobūdis arba kitaip – savybės, kurias galima nustatinėti bei vertinti naudojantis metodu. Metodai, kurių išeiga yra nefunkcinių veikimo savybių įverčiai (įvardinti šio darbo 1.1 skyriuje) – tinkami.
- Vertinimo rezultatų išraiškos pobūdis. T.y. rezultatai išreiškiami kokybiškai ar kiekybiškai. Siekiant kiek įmanoma didesnio objektyvumo, metodai, kurie pateikia rezultatus kiekybiškai yra labiau tinkami, nei tie, kurie pateikia kokybinius rezultatus. Tai suteikia pranašumo ir tarpusavyje lyginant gautus rezultatus su skirtingomis įeigomis.

- Reikalingo įdėti darbo bei laiko kiekis, norint gauti vertinimo rezultatus. Kuo darbo bei laiko reikia mažiau – tuo metodas yra naudingesnis, nes turi didesnę tikimybę būti naudojamas. Taip yra todėl, kad vis dar auga projektų atliekamų pagal Agile metodologiją procentinė dalis [SOA17], o kaip jau minėta anksčiau, dirbant pagal Agile metodologijas, architektūra yra kuriama kiekvienos iteracijos pradžioje, projektuojant tik tiek, kiek reikia iteracijai ir kuo mažesnėmis sąnaudomis;

Akivaizdu, kad pagal aukščiau įvardintus kriterijus, visi metodai, patenkantys į kai kurias iš kategorijų, nebus įvertinti teigiamai vieno ar kito kriterijaus atžvilgiu. Patirtimi grįsti vertinimai negali pasiūlyti kiekybinių vertinimo rezultatų, nes dėl ypatingai didelio kiekio galimų skirtingų situacijų projektuojant architektūrą, renkantis stilius, ekspertai neturi galimybės kiekybiškai įvertinti vieną ar kitą nefunkcinę savybę. Scenarijais grįstuose metoduose, įprastu atveju, įeiga – konkretus numatomos programų sistemos scenarijus (funkcinis reikalavimas). Architektūriniu požiūriu, jis gali būti įgyvendintas įvairiai, todėl tokio tipo metodai netinka siekiant darbo tikslo.

Iš visų metodų kategorijų, simuliacijomis ir matematiniais modeliais grįsti metodai galėtų būti teigiamai įvertinti pagal pasirinktus kriterijus. Simuliacijomis pagrįstų metodų poreikis pastebėtas gerokai anksčiau: Maria savo straipsnyje [Mar97] teigia, jog pastangų kiekis, reikalingas atlikti pakeitimus (pakeisti konfigūracijas) modelyje yra gerokai mažesnis, nei pastangų kiekis reikalingas atlikti pakeitimus realioje sistemoje – tai nepraktiška ir neefektyvu.

Sekančiuose skyriuose rasti metodai yra apžvelgiami ir įvertinami pagal kriterijus.

3.2.1. Reference Architecture Representation Environment (RARE) / Architecture Recovery, Change and Decay Evaluator (ARCADE)

RARE ir ARCADE yra dalis SEPA (angl. *Software Engineering Process Activities*) įrankių rinkinio. RARE naudojamas architektūros specifikavimui, o ARCADE simuliacija grįstam įvertinimui. Tikslas: architektūros specifikaciją parengus naudojant RARE, galėti įvertinti savybes naudojant ARCADE simuliaciją [Mår06].

RARE šiame darbe nėra aktualus, nes suteikia galimybę analizuoti tik statines savybes, priešingai, nei ARCADE. Barber et al. straipsnyje [BGH+02] nagrinėjamas ARCADE ir jame teigiama, jog jis automatizuoja korektiškumo, veikimo savybių (angl. *performance*) bei patikimumo vertinimą. Korektiškumas vertinamas tikrinant modelį, vykdant valdomą simuliaciją. Veikimo

savybės vertinamos remiantis diskrečių įvykių simuliacijomis (angl. *discrete event simulations*), o patikimumas – tikimybiniais grafų modelių (angl. *probabilistic graph model*) algoritmais. Šiame darbe aktualu tai, kaip ARCADE metode yra vertinamos veikimo savybės, nes jos vertinamos pasitelkiant simuliaciją.

Simuliacija ARCADE metode atliekama su Simpack įrankio pagalba. Turint sistemos naudojimo profilius (pvz. tokie panaudos atvejai, kaip: pasiekti produkto informaciją, atnaujinti produktą) bei jų kvietimo dažnius (nustatoma pagal tai, kiek dažnai funkcionalumas bus kviečiamas), pradėjus simuliaciją, atsitiktinai (pagal dažnį) yra parenkamas naudotojo profilis ir jis imamas vykdyti. Kiekvienas naudotojo profilis turi paslaugų (kiekviena iš jų turi kokybiškai apibūdintą atlikimo trukmę) eilę, kuriomis reikia pasinaudoti. Įvykdžius vieną profilį, imamas kitas, kol baigiasi simuliacijai skirtas laikas. Simuliacijos metu Simpack įrankis renka statistiką: paslaugų išnaudojamumą (angl. *utilization*), naudojimo profilio vykdymo trukmę (simuliaciniais laiko vienetais) ir kt. duomenis.

ARCADE metodas akivaizdžiai neatitinka vieno anksčiau išsikelto kriterijaus. Metodo įeigos pobūdis – konkretūs panaudos atvejai, o ne architektūrinių stilių ypatybės, todėl šis metodas netinka įgyvendinti darbo tikslą.

3.2.2. ARGUS-I

Tai yra specifikacijomis grįstas vertinimo metodas, kuris suteikia galimybę vertinti daugiau nei vieną architektūros savybę. Juo galima atlikti tiek architektūros struktūrinę analizę, tiek statinę bei dinaminę elgsenos analizę. Metodas naudoja formalius architektūros ir jos komponentų aprašymus – specifikacijas, kartu su komponentų būsenų diagramomis. Tuomet tas specifikacijas naudoja architektūros veikimo savybėms tirti, korektiškumui nustatyti [Mår06].

Vieira et al. straipsnyje [VDR00] analizuoja ARGUS-I metodo galimybes. Vienas iš analizės aspektų – simuliacija. Autorius teigia, jog ARGUS-I simuliacija leidžia aptikti defektus (pvz. būsenos pokytis turėtų iššaukti veiksmą, tačiau taip neatsitinka), vertinti veikimo savybes pagal tai, kiek dažnai yra iškviečiamas vienas ar kitas sistemos komponentas. Komponentai aprašomi būsenų diagramomis, o simuliacijos metu visi komponentų būsenų „varikliai“ veikia lygiagrečiai. Iš straipsnio [VDR00] galima daryti išvadą, jog metodo įeiga dalinai atitinka iškeltą kriterijų, nes įtraukia jungtis tarp komponentų, tačiau nėra minimi jokie galimi jungčių parametrai ar tinklo parametrai. Metodo išeiga gana skurdi atsižvelgiant į iškeltus kriterijus, nes pateikia tik komponentų iškvietimo dažnį. Taip pat,

metodas nėra populiarus (šaltinių, kuriuose jis būtų minimas ar naudojamas – labai nedaug), todėl apie jį ir jo galimybes objektyviai spręsti sudėtinga.

3.2.3. Layered Queuing Networks (LQN)

Abstraktus ir universalus modelis, galintis įvertinti įvairių tipų sistemas, neretai naudojamas programų sistemų veikimo savybėms nustatinėti. Metodas remiasi architektūros transformacija į LQN modelį. Modelis aprašo komponentų tarpusavio bendravimą, o jam sukonstruoti būtina žinoti komponentų elgsenos parametrus: kiek laiko užtrunka tam tikras veiksmas, kiek išteklių reikia jam atlikti [Mår06].

Mokomajame Woodside straipsnyje [Woo13] pateikiamos LQN modelio galimybės bei modelio metamodelis. Esminis principas – komponentų sluoksniavimas. T.y. gali būti ne vienas lygis, kuris užtikrina tam tikro funkcionalumo įgyvendinimą. Pavyzdžiui, HTTP serveris kreipiasi į programą, įgyvendinančią tam tikrą funkcionalumą, pastaroji – į duomenų bazę, o ji – į kietąjį diską. Sudarant modelį aktualios sąvokos: užduotis (angl. *task*), procesorius (angl. *host*), įrašas (angl. *entry),* pasikreipimas (angl. *call*) bei poreikis (angl. *demand*):

- Procesorius – esybė, skirta vykdyti operacijas. Taip pat turi eilę, kurioje talpinamos užduotys (ne užklausos!) bei eilės valdymo tvarką;
- Užduotis – sąveikaujanti modelio esybė, kuri turi savyje: užklausų eilę, eilės valdymo tvarką ir parametrą, kuris nurodo, kiek tokių egzempliorių veikia vienu metu (angl. *multiplicity*);
- Įrašas – esybė, kuri vaizduoja skirtingas operacijas, kurias užduotis gali atlikti. Įrašai turi „mąstymo“ laiką ir sąrašą pasikreipimų į kitus įrašus;
- Pasikreipimas – užklausa, atliekama iš vienos užduoties įrašo į kitos užduoties įrašą. Pasikreipimas gali būti sinchroniškas, asinchroniškas ar peradresavimo;
- Poreikis – bendra vidutinė suma reikalingo skaičiavimo laiko ir vidutinis pasikreipimų skaičius reikalingas įvykdyti įrašą.

LQN leidžia modeliuoti: tinklo užklausų vėlavimus, buferius, kritines sekcijas bei užraktus (angl. *lock*s). Galima modeliuoti ir naudotojų elgseną arba apkrovą (parametrai, tokie kaip laiko tarpas tarp užklausų ir pan.). Tačiau LQN neturi galimybės modeliuoti komponentų būsenas, todėl neatitinka vieno iš esminių kriterijų.

3.2.4. Scenario-based Architecture Reengineering (SBAR)

Šis metodas siūlo ne tik simuliacijomis, bet ir scenarijais grįstą vertinimą, tačiau šiame darbe aktualus tik šio metodo gebėjimas atlikti analizę simuliacijos būdu. Metodo autoriai savo straipsnyje [BB98] teigia, jog simuliacija įgyvendinama realizuojant pagrindinius architektūros komponentus, o taip pat, simuliuojant kontekstą, kuriame visa sistema turėtų veikti. Tai papildo metodą, nes leidžia vertinti veikimo savybes arba kitus nefunkcinius reikalavimus.

Darbo autoriui pavyko rasti labai nedaug šaltinių, kuriuose būtų minimas šis metodas. Taip pat, nei viename iš rastų šaltinių nėra tiksliai apibūdinta, kaip simuliacija yra vykdoma, todėl neįmanoma įvertinti, ar metodas atitinka kriterijus.

3.3. Agentais grįstų modelių simuliacijomis paremti vertinimo metodai

3.3.1. Programų sistemų architektūrų tyrimas taikant agentais grįstą modeliavimą

Vienas iš darbų, kuriuose programų sistemų architektūros modeliuojamos agentais, yra Mikoliūno magistrinis darbas [Mik14]. Esminė darbo dalis – agentais grįstas modelis, kuris tinka modeliuoti programų sistemas. Modelį sudaro:

- Komponentai. Darbe naudojama sąvoka „programinis komponentas“, kuri apibrėžiama kaip programų sistemų dalis, kuri vykdo duomenų apdorojimą, pateikia ir naudoja interfeisus. Tai gali būti biblioteka, modulis, klasė, priklausomai nuo pasirinkto detalumo. Komponentai modeliuojami kaip agentai, turintys atributus, tokius kaip maksimalus lygiagrečių užklausų vykdymo kiekis vienetais ar komponento vykdymo pasibaigimo klaida tikimybė;
- Komponentų sąryšiai. Sąryšiai, egzistuojantys tarp komponentų, kurie modeliuojami jungtimis. Galimi trijų tipų sąryšiai: asinchroninis, sinchroninis, sąryšis per tarpininką. Ryšiai gali būti vietiniai, kuomet kviečiantysis ir kviečiamasis yra tame pačiame serveryje ir nuotoliniai, kuomet yra skirtinguose serveriuose;
- Ištekliai. Serveris, kuriame veikia programų sistema modeliuojamas atskiru agentu, į kurį kiti agentai „kreipiasi“ norėdami atlikti užduotis ar siųsti užduotis kitiems komponentams. Galimi parametrai: procesoriaus išteklių vienetų kiekis, operatyviosios atminties kiekis, disko operacijų kiekis, serveriui prieinamo tinklo išteklio kiekis. Daroma prielaida, kad kiekvienai

komponento užduočiai atlikti reikia vienodo kiekio išteklių, kuriuos jis pasiima prieš atlikdamas užduotį ir atiduoda serveriui jau atlikęs.

- Aplinka. Ji atkuriama sukuriant agentus, kurie atspindėtų: išorines sistemas, naudotojus. Išorinių sistemų parametrai trys: atsako laikas (konstanta), klaidos tikimybė, planuoto neveikimo tikimybė. Naudotojai gali būti naudojami ir sistemos laiko įvykiams inicijuoti. Naudotojo agentas turi daug parametrų, skirtų nustatyti, kiek, kokiu laiko tarpu ir t.t. generuoti užklausas.

Vertinant Mikoliūno darbe aprašytą modelį pagal išsikeltus kriterijus, galima pastebėti jame esantį trūkumą, jog nėra modeliuojami tinklo parametrai (apžvelgti ankstesniame šio darbo skyriuje), nors pats modelis leidžia modeliuoti sistemas, kurių komponentai veikia ne viename serveryje, taip pat, nemodeliuojamos komponentų būsenos. Kiti pastebėti trūkumai:

- Nėra modeliuojamas siunčiamos užduoties dydis tarp komponentų, kas, priklausomai nuo tinklo pajėgumo, gali turėti įtaką;
- Siunčiant užduotį tarp komponentų ištekliai, reikalingi užduoties perdavimui, yra išskaičiuojami tik iš siuntėjo „sąskaitos“. Akivaizdu, jog siunčiant užduotį tarp komponentų esančių skirtinguose fiziniuose serveriuose, gavėjas gali būti išnaudojęs pvz. savo tinklo išteklius ir negalėti priimti užduoties;
- Netikslu modeliuoti, jog visos vieno komponento užduotys yra vienodo dydžio – reikalauja vienodo kiekio išteklių. Praktikoje taip nėra, o skirtumai tarp užduočių (išteklių sunaudojime) gali turėti įtakos simuliacijos rezultatams.

Modelio simuliacija šiame darbe vykdoma pasitelkus Repast Simphony karkasą. Sukurtas elementas „SimuliacijosValdiklis“, kuris inicijuoja agentų, sąryšių kūrimą, simuliacijos užbaigimą. Sąryšius generuoja „ArchitektūrosGeneratorius“, kuris sujungia turimus agentus pagal pateiktus parametrus. Simuliacijos vykdymą sudaro trys etapai: inicijavimas, vykdymas ir užbaigimas. Paskutiniame etape skaičiuojami nefunkcinių savybių įverčiai. Simuliacija vykdoma žingsniais: kiekvienam agentui yra iškviečiamas metodas, kuris įvykdo vieną agento žingsnį.

Metodika validuojama sukuriant realią sistemą ir tai sistemai sukuriant atitinkamą agentais grįstą modelį: lyginamos realios sistemos nefunkcinės savybės su gautomis savybėmis atliekant agentais grįsto modelio simuliaciją. Galima pastebėti, jog nėra validuojami nei asinchroniniai sąryšiai, nei galimybė modelyje turėti kelis serverius.

Naudojant sukurtą metodiką, darbe yra lyginami SOA ir EDA architektūriniai šablonai. Pateikiamos išvados, jog iš simuliacijos gautos šablonų savybės atitinka jau turimas žinias apie šiuos šablonus. Tačiau galima kritikuoti pasirinkimą lyginti būtent šiuos du architektūrinius stilius (šablonus) tarpusavyje, kurių paskirtis yra skirtinga. Tai įrodyti galima pasitelkiant SOA ir EDA apibrėžimus. SOA – būdas projektuoti programų sistemą taip, kad servisai būtų pasiekiami arba galutiniams naudotojams, arba kitiems servisams per viešus interfeisus. Servisas – verslo/veiklos funkciją atliekanti, į aiškų ir dokumentuotą interfeisą įvilkta, juodoji dėžė. Akcentuojama tai, jog funkcionalumas į servisus turi būti padalintas taip, kad leistų servisus pakartotinai panaudoti, kurti iš jų prasmingas konfigūracijas [Pap03]. Kitaip tariant, SOA principai nekalba apie tai, kokiu būdu servisai komunikuoja tarpusavyje (sinchroniškai, asinchroniškai ar net įvykiais [Sch11]). Kitokia situacija yra su EDA – šis architektūrinis stilius, apžvelgtas šio darbo 2.4.1 skyriuje, kalba būtent apie tai, jog komponentai komunikuoja įvykiais.

3.3.2. Programų sistemų, kuriose yra komponentų su būsenomis, architektūrų tyrimas taikant agentais grįstą modeliavimą

Siliūnas magistriniame darbe [Sil16] naudoja Mikoliūno aprašyta modelį, kuris jau buvo apžvelgtas šiame darbe. Kadangi Siliūno darbo tema apima komponentus su būsenomis, darbo autorius aprašo būdus, kaip galima modeliuoti būsenas agentais grįstu modeliu. Jas autorius siūlo modeliuoti kaip agento parametrą ar parametrų rinkinį, kurių reikšmės gali kisti nuo sistemos veiklos:

- Komponento protokolo būsena, vidinė būsena modeliuojama kaip agento parametras, priskyrimo bei konfigūracijos būsena – kaip statinis agento parametras;
- Sistemos globali būsena modeliuojama kaip agento, kuris turi ryšius su visais kitais agentais, parametras;
- Naudotojo sesijos bei naudotojo nuolatinė būsena modeliuojama kaip agento, kuris atspindi naudotoją, parametras.

Galima pastebėti, jog nėra esminio skirtumo tarp sesijos ir nuolatinės naudotojo būsenos: nenagrinėjamas atvejis, kuomet naudotojas gali atsijungti/prisijungti iš naujo, taip įgydamas naują sesiją ir prarasdamas buvusius parametrus.

Tiek modelio simuliacija, tiek modelio validacija vykdoma be esminių skirtumų nuo Mikoliūno pasiūlyto varianto. Naudojant sukurtą metodiką taip pat lyginami EDA ir SOA

architektūriniai stiliai. Atsižvelgiant į išsikeltus kriterijus, ši metodika neatsižvelgia tik į tinklo parametrus.

3.4. Išvados

Nei vienas iš apžvelgtų simuliacinių architektūrų savybių vertinimo metodų pagal išsikeltus kriterijus negali pasiūlyti to, ką siūlo agentiniai Mikoliūno bei Siliūno modeliai: dalis jų tinka tik konkrečios architektūros, o ne architektūrinių stilių analizei, kiti, tokie kaip LQN, neturi tiek parametrų ir galimybių (pvz. būsenų), kad būtų galima simuliuoti ir vertinti tinkle išskirstytų programų sistemų architektūrinių stilių savybes. Taip pat, SBAR ir ARGUS-I metodai yra nepopuliarūs ir aktyviai nebetobulinami, nes straipsnių ar kitos literatūros susijusios su šiais metodais – nedaug.

4. MODELIAVIMAS AGENTAIS

4.1. Agentais grįsto modelio apibrėžimas

Kadangi agentais grįsto modelio sudarymui reikia agentų, pradžioje reikėtų apibrėžti, kas tai yra agentas. Įvairiuose šaltiniuose, agentas apibrėžiamas per jo turimas savybes. Straipsnyje [MN09] agentams priskiriamos septynios savybės. Agentai yra:

- Autonomiški. Agentas gali funkcionuoti nepriklausomai nuo aplinkos ar sąveikavimo su kitais agentais;
- Moduliarūs. Galima nustatyti, ar kažkas (modelio būsenos elementas) priklauso agentui, ar ne.
- Socialūs. Agentas turi protokolus ar mechanizmus, kurie apibūdina, kaip jis bendrauja su kitais agentais;
- Galimai priklausomi nuo aplinkos. Agentas gali sąveikauti ir su aplinka, kurioje jis yra. T.y. jo veikimas gali priklausyti nuo aplinkos, kurioje jis yra patalpintas;
- Galimai turintys tikslus. Siekdamas tikslų agentas gali modifikuoti savo veiklą taip, kad tikslai būtų pasiekti ar prie jų būtų priartėta;
- Galimai besimokantys ir sugebantys adaptuotis. Agentas gali koreguoti savo veiklą pagal praeities įvykius;
- Galimai turintys atributus. Agentas gali turėti atributus, kurie nusakytų, jog agentui trūksta tam tikrų išteklių ar pan.

Agentais grįstas modelis – rinkinys agentų ir jų tarpusavio sąryšių. Jis sudaromas aprašant sistemos sudėtines dalis, todėl tai yra žmogui natūralus būdas sudaryti modelį, priešingai nei sistemos elgsenos aprašinėjimas matematinėmis lygtimis [Bon02].

Esminis agentais grįsto modelio aspektas – ryšiai tarp agentų. Straipsnyje [MN09] pateikiamos topologijos, kurios nusako agentų sąryšių galimas schemas:

- Neerdvinė schema. Agentai neturi lokacijos, o pats modelis neturi erdvinio atvaizdavimo;
- Tinklelis. Atvaizduoja agentų ryšius kaip tinklelį. Aplink agentą esantys kiti agentai – kaimynai. Agentai gali keisti savo padėtį judėdami tinkleliu;
- Euklidinė erdvė. Agentai vaizduojami 2D, 3D ar erdvėje su daugiau dimensijų;
- GIS. Geografinių informacinių sistemų schema. Agentai gali judėti realistišku geoerdviniu paviršiumi;
- Tinklas. Gali būti dinaminis ar statinis. Dinaminis tinklas gali keisti ryšius, statinis – ne.

Modelio kūrimo eiga, įrankiai modeliavimui nagrinėjami Siliūno darbe [Sil16]. Šiame darbe bus laikomasi tokios pat eigos bei pasirinktas tas pats įrankis modeliavimui: „Repast Simphony".

4.2. Agentais grįstų modelių verifikacija ir validacija

Simuliacijos modelis įprastu atveju yra tikros sistemos abstrakcija, kuri leidžia mums geriau suprasti ir nuspėti sistemos veikimą, tačiau abstrakcijos ir prielaidos gali iškreipti modelį ir padaryti jį netiksliu. Todėl naudojantis metodika ar metodu, kurio esminis principas yra simuliacija, viena iš svarbiausių užduočių – nustatyti, kiek sistemos modelis yra tikslus, lyginant jį su realia sistema [XKM05]. Tai galima padaryti verifikuojant ir validuojant.

Knygoje [BF14] pateikiami apibrėžimai:

- Verifikacija – bandymas užtikrinti, jog produktas yra sukurtas korektiškai: veiklų išeigos atitinka apibrėžtas specifikacijas;
- Validacija – bandymas užtikrinti, jog sukurtas produktas tinkamai atlieka savo paskirtį.

Abu šie procesai yra modelio kūrimo proceso veiklos. Jų metu modelis yra tobulinamas tol, kol jo tikslumas ima tenkinti.

Yra ne vienas metodas, kaip galima validuoti ir verifikuoti agentais grįstą modelį. Straipsnyje [XKM05] yra pateikiami tokie būdai:

- Išorės validavimas (angl. *face validity*). Klausiama srities ekspertų, ar jų nuomone modelis yra pakankamai tikslus. Yra du būdai, kaip galima ekspertams pateikti modelį:
    - Animacija. Vaizdinis modelio elgsenos atvaizdavimas laike. Pačios simuliacijos metu galima stebėti parametrų reikšmes ir pan.;
    - Grafinė reprezentacija. Pateikiami modelio išeigos parametrai, grafikų pavidalu.
- Sekimas (angl. *tracing*). Būdas panašus į validavimą naudojant animaciją. Sekama modelio elgsena;
- Vidinė validacija (angl. *internal validity*). Simuliuojama keletą kartų su skirtingais atsitiktinių skaičių generatorių pradiniais duomenimis. Jei tai sukelia didelius skirtumus išeigoje – modelis galimai nevalidus;
- Istorinių duomenų validacija (angl. *historical data validation*). Imami realios sistemos istoriniai duomenys (jei tokie egzistuoja). Tuomet pagal dalį jų sudaromas modelis. Su likusia dalimi modelis yra validuojamas, ar elgiasi taip, kaip turėtų;
- Parametro kintamumo-jautrumo analizė (angl. *parameter variability-sensitivity analysis*). Keičiamas modelio įeigos parametras ir stebima, kiek pasikeičia išeigos parametrai. Pokytis turėtų būti toks pats, kaip realioje sistemoje pakeitus tą patį įeigos parametrą;
- Tiuringo testas (angl. *Turing test*). Sistemos ekspertams duodamos išeigų reikšmės tiek realios sistemos, tiek modelio simuliacijos. Tuomet klausiama, ar ekspertai gali atskirti, kur yra realios sistemos reikšmės, o kur – modelio;
- Modelių palyginimas (angl. *model-to-model comparison*). Paimami tos pačios sistemos skirtingi modeliai – modeliai kurti kitų komandų, kitais įrankiais ir pan. Skirtumai tarp išeigos parametrų gali atskleisti modelio netikslumus.

Norint tiksliai atlikti validaciją, galima naudoti statistinius metodus kartu su aukščiau išvardintais validacijos būdais.

Kadangi agentais grįstų modelių simuliacijos gali būti sudėtingos dėl daug tarpusavyje, vienu metu, komunikuojančių agentų ar daugybės jų parametrų, nėra lengva juos validuoti. Tą patį teigia ir Niazi et al. straipsnyje [NHK09], kuriame siūlomas agentais grįstų modelių validavimo metodas VOMAS. Straipsnio autorius teigia, kad VOMAS (angl. *The Virtual Overlay Multi-agent System*) leidžia validuoti:

- Erdviškai – agentų išsidėstymą modelyje;
- Neerdviškai – invariantus, kompleksinius duomenis;

- Sąryšius – galimas variantas, jog agentų tarpusavio sąryšiai yra svarbiau, nei agentų išsidėstymas modelyje;
- Atstumą tarp agentų (angl. *proximity based validation*) – tam tikruose modeliuose svarbu sekti situaciją, kiek vienas agentas yra nutolęs nuo kito. Pvz. aukos-plėšrūno modeliuose;
- Žurnalo įrašus (angl. *log based validation*) – prieš paleidžiant simuliaciją nustačius vietas, kurios turėtų pildyti žurnalą įrašais, po to jį galima analizuoti;
- Invariantus – modelyje gali būti reikšmių, kurios niekuomet neturi būti peržengtos. Jei taip atsitinka – stebėtojui turi būti pranešama.

Šis validavimo metodas veikia sukuriant multiagentinę sistemą tam agentiniam modeliui, kurį norima validuoti. Straipsnyje pateikiama veiksmų eiga, kaip reikia naudotis metodu, taip pat, pateikiama metodo sandara.

LITERATŪROS APŽVALGOS IŠVADOS

Šio darbo literatūros apžvalgoje nagrinėtos savybės, kuriomis gali pasižymėti programų sistemos, tinklai ir išskirtinai tos programų sistemos, kurios yra išskirstytos tinkle. Tai yra būtina tam, kad metodo kūrimas, tyrimas būtų kryptingi, nes savybės yra aspektai, kuriais galima programų sistemas vertinti. Nors šio darbo tyrimo objektas yra architektūriniai stiliai, pastarieji kaip tik ir suteikia programų sistemoms tam tikras savybes. Programų sistemų savybių apžvalgai buvo pasitelktas ISO 25010 modelis ir detaliau apžvelgtos savybės, kurias galima vertinti atliekant simuliacijas. Kadangi darbo tema susijusi su išskirstytomis programų sistemomis tinkle, aktualiausia tirti yra tas specifines savybes, kuriomis pasižymi tokio tipo sistemos. Dėl šios priežasties, apžvelgta CAP teorema bei jos patikslinimas PACELC, kurios orientuotos į savybes pasireiškiančias tuomet, kai išskirstytos programų sistemos komponentai turi bendrą būseną. Apibrėžta ir būsenos sąvoka bei išryškinta būsenų kategorija, aktuali šiam darbui – būsenos, kurios yra bendros tarp pasirinktų komponentų.

Literatūros apžvalgoje apibrėžtas ir išnagrinėtas pats tyrimo objektas – architektūriniai stiliai. Išanalizuoti architektūros, architektūrinių stilių apibrėžimai rasti įvairiuose šaltiniuose. Taip pat, architektūriniai stiliai kategorizuoti pagal tai, kam jie skirti. Šiame darbe pasirinkta tirti komunikacijos tipo stilius, todėl du iš jų (įvykiais ir užklausomis grįsti) nagrinėti detaliau.

Kadangi šio darbo tikslas – sukurti metodą, apžvelgta metodų kategorizacija. Taip pat, atlikta ir simuliacinių metodų, skirtų programų sistemų savybėms nustatinėti, apžvalga, vertinimas pagal pasirinktus kriterijus. Analizuoti du magistriniai darbai, programų sistemų savybėms nustatinėti naudojantys agentais grįstą simuliaciją. Įžvelgti magistriniuose darbuose naudojamos metodikos trūkumai, kuriuos reikėtų pašalinti šiame darbe. Apžvelgti ir alternatyvūs ARCADE, ARGUS-I, LQN, SBAR metodai. Įvertinus atitinkamumą kriterijams, prieita išvada, jog agentais grįsta simuliacija šiame darbe labiausiai tinka analizuoti architektūrinių stilių programų sistemoms suteikiamas savybes.

Šiame darbe kuriamas metodas bus paremtas agentais grįstu modeliu ir jo simuliacija. Todėl literatūros apžvalgoje pateikti agento, agentinio modelio apibrėžimai. Pagrindžiant metodo teisingumą, būtina atlikti validaciją ar/ir verifikaciją. Dėl šios priežasties analizuoti galimi validacijos ir verifikacijos metodai bei detaliau apžvelgtas validacijos metodas VOMAS.

5. ARCHITEKTŪRINIŲ STILIŲ BEI SPRENDIMŲ MODELIAVIMAS AGENTAIS

Šiame skyriuje bus kuriami nurodymai bei gairės, siekiant sukurti metodą, kuris leistų modeliuoti architektūrinius stilius bei sprendimus. Bus remiamasi Mikoliūno [Mik14] bei Siliūno [Sil16] darbuose aprašytomis metodikomis, kurios buvo apžvelgtos ankstesniuose šio darbo skyriuose. Vienas iš pagrindinių skirtumų tarp minėtų metodikų bei šiame darbe kuriamo metodo yra tai, jog šiame darbe akcentuojamas stilių bei sprendimų, o ne programų sistemų ar jų dalių modeliavimas. Iš to seka, jog šis metodas nėra skirtas programų sistemų architektūrų lyginimui. Realios programų sistemos turi pernelyg daug komponentų, kai kurių komponentų įtaka visai programų sistemai gali būti nykstamai maža ir pan. Taip pat, modeliuojant visą programų sistemą vienas iš 3.2 skyriuje aprašytų kriterijų (jog norint pasinaudoti metodu neturi reikėti daug darbo bei laiko) būtų nebetenkinamas.

Prieš modeliuojant architektūrinius elementus agentais, būtina susieti, koks architektūrinis elementas, kokį agentinio modelio elementą atitinka. Tokį susiejimą galima rasti Siliūno magistriniame darbe [Sil16]:

Architektūros elementai	**Modelio elementai**
Komponentas	Agentas
Komponento veiklos modelis	Agento elgsena
Jungimo mechanizmas	Agentų ryšys
Vykdymo ištekliai	Agentų aplinka
Sistemos aplinka	Agentų aplinka

1 lent.: Ryšys tarp konceptualių architektūros elementų ir agentais grįsto modelio elementų.

Siliūno darbe pateikiami ir architektūros elementų apibrėžimai. Komponentas – abstrakti esybė atliekanti jai pavestas funkcijas. Komponento veiklos modelis – taisyklių ir funkcijų rinkinys įtraukiantis užklausos apdorojimo veiksmų aprašą, komponento būseną, bei apibrėžiantis veiklos priklausomybę nuo būsenos. Jungimo mechanizmas – elementas, kuris apibrėžia ryšį tarp komponentų. Sistemos aplinka – elementas nusakantis kaip programų sistema vykdymo metu yra susijusi su aplinka kurioje egzistuoja. Vykdymo ištekliai – komponentų veiklą ribojantys ištekliai.

Kadangi pats modeliavimas ir simuliavimas būtų beprasmis, jei nebūtų galima surinkti duomenų, kurie atspindėtų architektūrinio stiliaus ar sprendimo suteikiamas savybes, todėl privalu

metode numatyti galimybę rinkti tokio tipo duomenis. Siliūno darbe pateikiamoje metodikoje komponento veiklos modelis susiejamas su kokybės atributo reikšme, o pastaroji – su kokybės matu. Tai reiškia, jog komponento elgseną galima vertinti pasirinkus tam tikrus atributus (galimi variantai yra apibūdinti 1.1 skyriuje), kurių kiekvienas gali turėti daugiau nei vieną matą. Iš to seka, jog viso sumodeliuoto stiliaus ar sprendimo vertinimas pagal pasirinktą atributą atliekamas po simuliacijos, atsižvelgiant į kiekvieno modelyje esančio komponento surinktą statistiką pagal to atributo matus.

5.1. Modelis

Žemiau bus pateikiamos taisyklės, kaip yra kuriamas agentinis modelis arba kitaip – kaip susiejami architektūriniai elementai su agentinio modelio elementais. Pagrinde bus naudojamos Siliūno darbe aprašytos modeliavimo taisyklės jas pakeičiant, papildant ar pašalinant taip, kad metodas leistų įgyvendinti darbo tikslus: metode turėtų atsirasti galimybė modeliuoti tinklo elgseną bei galimybė modeliuoti bendrą būseną tarp kelių komponentų (sujungtų į vieną tinklą kompiuterių grupės modeliavimas).

5.1.1. Komponentai

Komponento apimtis modelyje gali būti įvairi (pvz. nuo vieno objektinės paradigmos metodo iki visos sistemos), todėl tai nusprendžia metodo naudotojas modelio kūrimo metu. Tikslas – atkurti architektūrinį stilių arba sprendimą. Tai reiškia, jog komponento mastelis (angl. *scale*) turi būti kiek įmanoma stambesnis (tokiu atveju, reikia įdėti mažiau darbo kuriant modelį), tačiau pakankamai smulkus, kad stilius ar sprendimas modelyje būtų atkurtas.

Komponentas modelyje turi galimybę apdoroti lygiagrečiai keletą užklausų, nes praktikoje lygiagretus užduočių atlikimas yra paplitęs. Gijų skaičius praktikoje visuomet yra apribotas, todėl ir modelyje galima nustatyti maksimalų lygiagrečiai atliekamų užduočių skaičių.

Atributų sąrašas [Sil16]:

- K_k – kviečiamų komponentų kvietimo tipas: *VISUS* – kviesti visus sujungtus komponentus, *ATSITIKTINIS* – kviesti atsitiktinį skaičių komponentų, tačiau maksimaliai K_{kmax}, minimaliai $K_{kmin;}$
- K_{kmax} – maksimalus kviečiamų komponentų skaičius (Jei K_k = *ATSITIKTINIS*. Ne didesnis už vaikinių komponentų skaičių);
- K_{kmin} – minimalus kviečiamų komponentų skaičius (Jei K_k = *ATSITIKTINIS*);

- K_l – maksimalus lygiagrečiai apdorojamų užklausų kiekis komponente. Nurodžius 0, lygiagrečių užklausų kiekis neribojamas;
- K_e – klaidos tikimybė apdorojant užklausą. Reikšmė nurodoma intervale [0;1];
- K_{dw} – tikimybė, jog komponentas yra išjungtas dėl atliekamų priežiūros darbų, arba tiesiog yra sugedęs. Reikšmė nurodoma intervale [0;1].

5.1.2. Ryšiai

Jungimo mechanizmo elementas modeliuojamas kaip ryšio tipo atributas agente. Kadangi agentas gali turėti daugiau ryšių su kitais agentais nei vieną, todėl agente saugomas ryšių rinkinys. Ryšys turi tokius atributus:

- R_{si} – siunčiamos informacijos kiekis, informacijos kiekio vienetais;
- R_{gi} – gaunamos informacijos kiekis, informacijos kiekio vienetais;
- R_t – ryšio tipas. Sinchroninis arba asinchroninis;
- R_{gk} – nuoroda į užklausos gavėją, komponentą;
- R_{sk} – nuoroda į užklausos siuntėją, komponentą;
- R_{dp} – ryšio duomenų perdavimo patikimumas. Tikimybė, jog tinklas suges per vieną laiko vienetą intervale [0;1];
- R_{dpa} – ryšio duomenų perdavimo atstatomumas. Laiko kiekis vienetais, kurio reikia, kad duomenys vėl galėtų būti perduodami ryšiu;
- R_b – ryšio būsena. *SVEIKAS* arba *SUGEDĘS*. Jei ryšys sveikas, juo gali būti perduodama užklausa. Jei sugedęs – negali.

Pagal ryšio atributus galima pastebėti, kad ryšys gali būti sinchroninis arba asinchroninis. Šiame darbe šie ryšių tipai bus modeliuojami tiksliau, nei Siliūno ar Mikoliūno darbuose. Sinchroninis tipas, kuomet išsiuntus užklausą yra laukiama, kol komponentas, kuriam užklausa yra išsiųsta atsakys. Laikas, reikalingas pilnai įvykdyti tokią užklausą susideda iš jos išsiuntimo, laukimo (kol užduotis įvykdoma) bei atsako gavimo. Į užduoties įvykdymo apibrėžimą įeina ir sinchroninių bei asinchroninių vaikinių užklausų išsiuntimas. T.y. jei komponentas A kreipiasi į komponentą B sinchroniškai, tuomet A lauks kol užklausa bus išsiųsta, taip pat, kol B įvykdys savo užduotį ir išsiųs užklausas kitiems komponentams bei kol grįš atsakas iš komponento B. Asinchroninis, kuomet išsiuntus užklausą nėra laukiama, kol bus gautas atsakymas. Laikas reikalingas tokiai užklausai atlikti susideda tik iš jos išsiuntimo.

Taip pat, ryšio atributai buvo papildyti ryšio būsena. Kadangi realaus pasaulio tinkluose gedimai nėra išvengiami ir tai įprastu atveju turi didelę įtaką visai programų sistemai ir jos elgsenai, vadinasi, toks atributas yra būtinas. Jei ryšys nutrūksta (ir būna nutrūkęs R_{dpa} laiko kiekį) užklausų siuntimo ar atsako gavimo metu, vadinasi tos užklausos pažymimos kaip nepavykusios.

Papildomi atributai agentams, kurie vaizduoja komponentus [Sil16]:

- K_r – ryšių rinkinys, kuris nurodo, į kuriuos kitus komponentus gali kreiptis komponentas.

5.1.3. Ištekliai

Mikoliūno bei Siliūno darbuose pasirinkta išteklius, reikalingus komponentų užduočių vykdymui, modeliuoti kaip atskirą serverį (atskirą agentą), į kurį yra įdiegiami komponentai. Šiame darbe pasirenkamas toks pat sprendimas. Tiek komponentų sukurtos užduotys, tiek užklausų bei atsakymų į užklausas siuntimas tarp komponentų yra atliekamos serverio. Komponentai pateikia norima atliktį veiksmą serveriui, o serveris pagal savo turimus išteklius tuos veiksmus vykdo. Siunčiant užklausas arba atsakus tarp komponentų, kurie yra įdiegti į skirtingus serverius, tinklo ištekliai yra naudojami tiek siunčiančiajame, tiek gaunančiajame serveryje, kitu atveju, tinklo ištekliai nėra naudojami.

Serverio agento atributai [Sil16]:

- S_p – procesoriaus išteklio kiekis;
- S_a – operatyviosios atminties išteklio kiekis;
- S_d – disko operacijų išteklio kiekis;
- S_s – užduočių serveryje atlikimo strategija: *EILĖ* – eilė, *ATVIRKŠTINĖ_EILĖ* – atvirkštinė eilė, *ATSITIKTINĖ* – atsitiktinė tvarka;
- S_{ta} – atsiunčiamų duomenų perdavimo išteklio kiekis (informacijos kiekio vienetai per laiko tarpą);
- S_{ti} – išsiunčiamų duomenų perdavimo išteklio kiekis (informacijos kiekio vienetai per laiko tarpą).

Reikalingi ištekliai atlikti užduotį yra saugomi kaip papildomi atributai komponento agente [Sil16]:

- K_s – nuoroda į serverį, kuriame komponentas yra įdiegtas;

- K_{cpu} – procesoriaus išteklių kiekis, reikalingas atlikti užklausos apdorojimo užduotis;
- K_o – operatyviosios atminties išteklių kiekis, reikalingas atlikti užklausos apdorojimo užduotis;
- K_d – disko operacijų išteklių kiekis, reikalingas atlikti užklausos apdorojimo užduotis;
- K_t – užduočiai atlikti reikalingas vykdymo laikas laiko vienetais.

Atributas, kuris nurodo, kiek procentiškai tinklo išteklių yra sunaudojama užklausos perdavimo metu šiame metode buvo panaikintas. Metodas yra patobulinamas taip, jog laikas, reikalingas užklausos perdavimui ir užklausos atsako grąžinimui yra skaičiuojamas pagal tai, kokio dydžio yra užklausa/atsakas (šie atributų reikšmės nurodomos ryšio atribute) dalinant jį iš duomenų perdavimo išteklio kiekio laiko tarpui. Jei perduodamos vienu metu daugiau nei viena užklausa/atsakas, duomenų perdavimo išteklio kiekis dalinamas po lygiai. Toks modeliavimo būdas yra pranašesnis nes tiksliau atspindi realią situaciją išskirstytose tinkle sistemose bei leidžia tiksliau sumodeliuoti sinchroninį bei asinchroninį perdavimą.

5.1.4. Aplinka

Kitaip nei Siliūno bei Mikoliūno darbuose, šiame darbe į modeliuojamos sistemos aplinkos apibrėžimą įeis tik naudotojai. Naudotojai plačiąja prasme – bet kas, kas gali generuoti (siųsti) užklausas. Išorinių sistemų apibrėžimo nutarta neįtraukti į aplinką, nes joms modeliuoti nėra būtinas atskiras agento apibrėžimas, jas galima sumodeliuoti serverio ir komponento agentais. Naudotojo atributų sąrašas [Sil16]:

- V_r – atributas, nurodantis ryšį, kuriuo yra siunčiamos užklausos. Šis ryšys neturi galimybės sugesti, kad būtų užtikrinamas stabilus užklausų tiekimas modeliuojamai sistemai (kad vertinant stiliaus ar sprendimo savybes nereikėtų atsižvelgti į šį ryšį ir jo elgseną);
- V_s – užklausų generavimo strategija: *ATSITIKTINĖ* – generuojama atsitiktinai, *REGULIARI* – generuojama reguliariai;
- V_a – užklausų sugeneravimo tikimybė per vieną laiko vienetą. Naudojama jei V_s = *ATSITIKTINĖ*. Sugeneruojama užklausų ne daugiau nei V_{umax} ir ne mažiau V_{umin}. Reikšmė nurodoma intervale [0;1];
- V_t – laiko vienetų kiekis tarp užklausų generavimo. Naudojamas jei V_s = *REGULIARI*. Sugeneruojamas užklausų kiekis ne didesnis nei V_{umax} ir ne mažesnis nei V_{umin};
- V_{umax} – maksimalus sugeneruojamų užklausų kiekis per užklausų generavimo laiko vienetą;

- V_{umin} – minimalus sugeneruojamų užklausų kiekis per užklausų generavimo laiko vienetą;
- V_k – reagavimo į nepavykusią užklausą strategija. *NEKARTOTI* – užklausos nekartoti, *KARTOTI* – kartoti užklausą;
- V_{uk} – maksimalus užklausų kartojimų kiekis. Jei reikšmė 0 – kartojama neribotą kiekį. Naudojamas jei V_k = *KARTOTI*;
- V_{ul} – laikas laiko vienetais, per kurį neišsiuntus užklausos ji yra pažymima kaip nepavykusi.

5.1.5. Būsenos

Šiame darbe, modeliuojant bendras būsenas tarp komponentų pasirinkta kiek kitokia strategija būsenai modeliuoti, nei Siliūno darbe. Pagrindinė to priežastis yra ta, jog šiame darbe aktuali ne pati būsena (jos struktūra ar pan.), o jos bendrinimo tarp komponentų eiga.

Keletą komponentų apimančios būsenos modeliavimas. Tokia būsena modeliuojama pasitelkiant užklausas, kurios keičia būseną. Modeliuojant turi būti nusprendžiama, kuriems modeliuojamo sprendimo komponentams būsena turi būti bendra. Laikoma, jog būsena keičiama kuomet komponentas gauna užklausą, kuri ją keičia. Būsenos bendrinimas vyksta taip: gavęs užklausą (kuri keičia būseną) iš sistemos naudotojo, komponentas, turėdamas sąrašą kitų komponentų, su kuriais būsena yra bendra, siunčia jiems užklausas (sinchroniniu arba asinchroniniu būdu), kad būtų atnaujinta jų būsena. Komponentas gavęs būsenos bendrinimo užklausą iš kito komponento siunčia užklausas kitiems, kuriuos turi sąraše išskyrus tam, iš kurio gavo. Taip pat, komponentas gali vienu metu apdoroti tik vieną užklausą, kuri keičia būseną. Likusios užklausos gautos tuo pačiu metu yra atmetamos.

Papildomi atributai komponente:

- K_{sh} – ryšių rinkinys, kuris nurodo, su kuriais kitais komponentais būsena yra bendrinama;
- K_{nvn} – atributas, kuris nurodo, ar užklausą laikyti nepavykusią, jei bent viena iš vaikinių užklausų nepavyko.

Papildomi atributai ryšyje:

- R_{sc} – atributas, kuris nurodo, ar siunčiamos užklausos keičia būseną komponente į kurį kreipiasi. *NEKEIČIA* – jei nekeičia, *KEIČIA* – jei keičia.

Siekiant neprarasti informacijos apie ryšius tarp tėvinių ir vaikinių būsenos bendrinimo užklausų, pasitelkiamas unikalus užklausos identifikatorius. Kiekviena užklausa jį gauna naują ir unikalų, išskyrus būsenos bendrinimo užklausas. Jų unikalus užklausos identifikatorius sutampa su tėvinės užklausos identifikatoriumi. T.y. siekiama modeliuoti taip, jog tuo atveju, kuomet sistemoje užklausų, kurios bendrina tą pačią būseną yra daugiau nei viena, būtų galima pagal unikalų identifikatorių grupuoti užklausas, kurios bendrina tą pačią būseną.

Taip pat, būsenos bendrinimo užklausos skiriasi nuo įprastų užklausų tuo, jog jų apdorojimo metu komponentai neatlieka užduočių ir nesiunčia užklausų kitiems komponentams, o siunčia tik būsenos bendrinimo užklausas (t.y. užklausos siunčiamos tik tais ryšiais, kurie yra atribute $K_{sh)}$. Toks sprendimas priimtas todėl, kad būsenos bendrinimo veiksmas komponente, po to, kai šis priima būsenos bendrinimo užklausą, kainuoja minimaliai išteklių ir laiko, todėl nedaro įtakos.

Papildomas atributas, kuris nurodo, ar užklausą laikyti nepavykusia, jei nepavyko kažkuri iš vaikinių užduočių reikalingas tam, kad būtų galima atkurti skirtingas būsenų bendrinimo strategijas. CAP teoremoje tai būtų esminis skirtumas CP arba AP sistemų. Trumpai apibūdinant, CP sistemos neleidžia esant tinklo trūkiams keisti būsenų, todėl užklausos būna nesėkmingos, o AP sistemos leidžia tai atlikti, tik tam tikri komponentai lieka su senesne būsena.

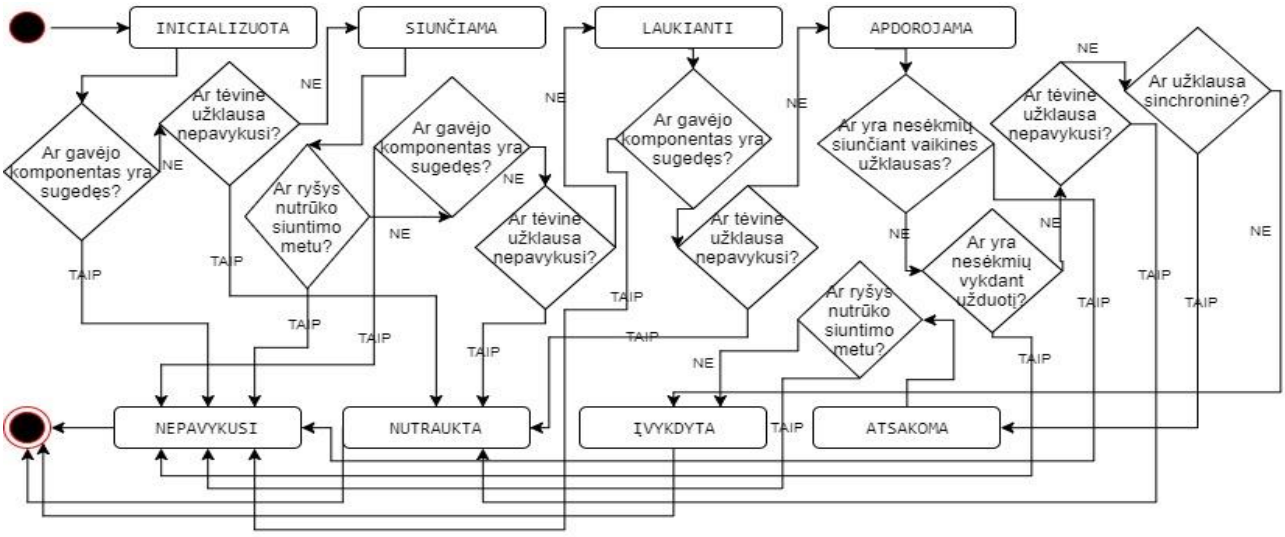
5.2. Modelio elementų galimų būsenų bei veiklos diagramos

Šiame skyriuje pateikiamos diagramos, kuriose pateiktų tėkmių privalu laikytis modeliuojant architektūrinius sprendimus vadovaujantis metodu. Užklausa šiame metode yra ypatingai svarbus elementas, nes didžioji dalis nefunkcinių savybių yra susijusios būtent su ja. Užklausa šiame metode gali turėti 8 būsenas:

- INICIALIZUOTA. Reiškia, jog užklausa buvo ką tik suformuota, tačiau su ja neatlikti jokie veiksmai;
- SIUNČIAMA. Reiškia, jog užklausa keliauja iš siuntėjo pas gavėją apibrėžtu ryšiu;
- LAUKIANTI. Reiškia, jog užklausa laukia, kol komponentas galės ją apdoroti. Laukimas gali tęstis tol, kol pvz. atsilaisvins komponento gija;
- APDOROJAMA. Reiškia, jog užklausa yra apdorojama komponento. Siunčiamos vaikinės užklausos, apdorojama užduotis;
- NEPAVYKUSI. Dėl įvairių priežasčių nepavykusi įvykdyti užklausa;

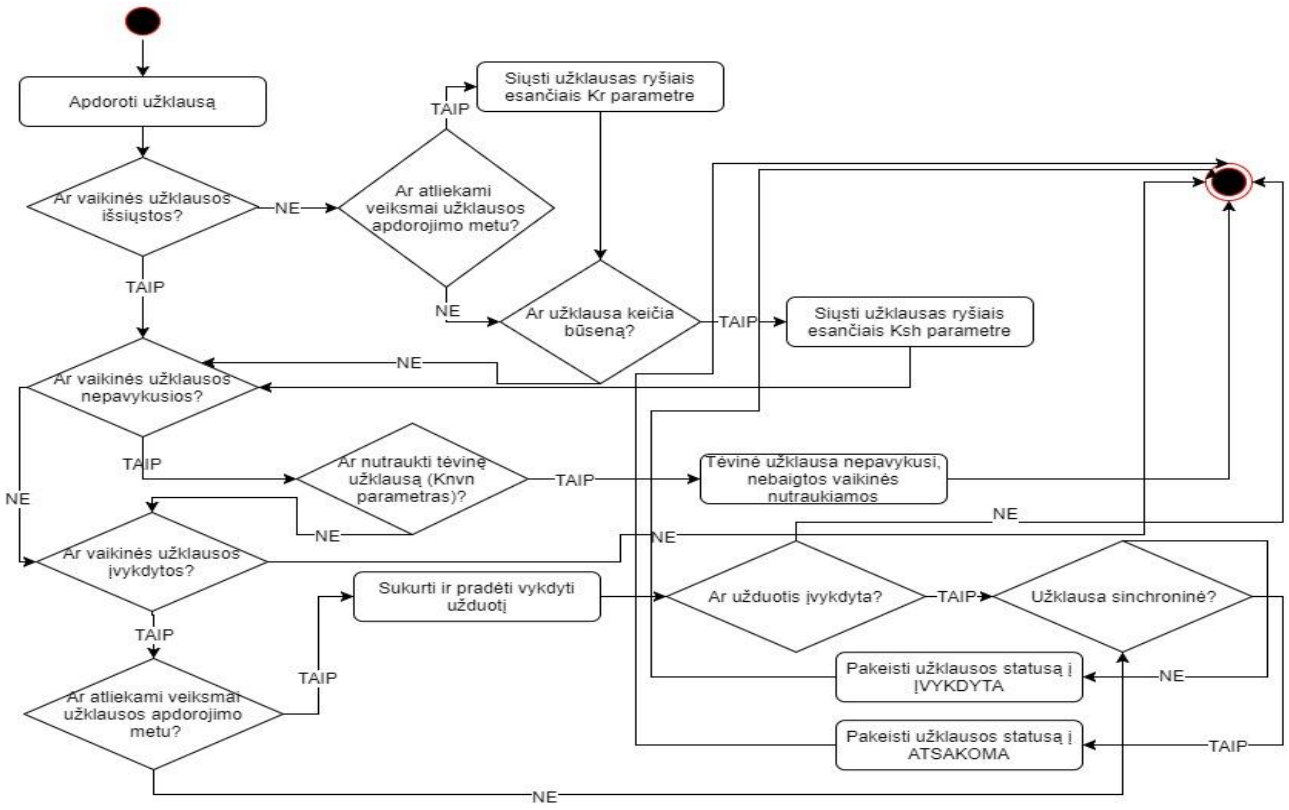
- NUTRAUKTA. Užklausa gali būti nutraukiama, jei nebėra prasmės ją vykdyti toliau. Pvz. jei viena iš vaikinių tėvinės užklausos užklausų nepavyko, nebėra prasmės vykdyti likusias vaikines užduotis, nes tėvinė vis vien bus nepavykusi;
- ĮVYKDYTA. Sėkmingai įvykdyta užklausa;
- ATSAKOMA. Reiškia, jog yra siunčiamas užklausos atsakas.

Žemiau pateikiama užklausos būsenos būsenų diagrama. T.y. vadovaujantis šia diagrama, galima nustatyti, kaip ir kokiais atvejais užklausa gali įgyti tam tikrą būseną:



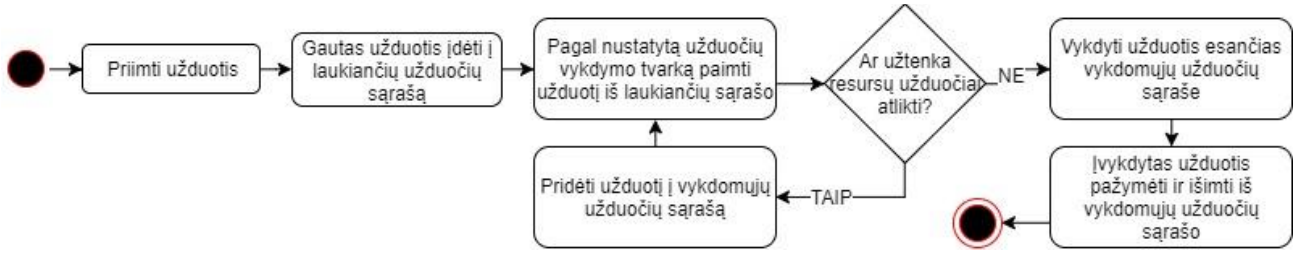
2 pav.: Užklausos būsenos būsenų diagrama.

Kitas svarbus aspektas šiame metode yra tai, kaip ir kokia tvarka modeliuojami komponentai atlieka veiksmus su gaunamomis užklausomis. Nuo to priklauso, kokie nefunkcinių savybių įverčiai bus gauti atliekant simuliaciją. Žemiau pateikiama komponentų veiksmų apdorojant užklausą veiklos diagrama. Į ją įtraukiami veiksmai, kurie atliekami pradėjus apdoroti užklausą iki kol yra pradedamas siųsti užklausos atsakas arba asinchroninės užklausos atveju, užklausa pažymima kaip įvykdyta. Būtina pastebėti, jog sumodeliuoto architektūrinio sprendimo simuliacijos vykdymo metu šią diagramoje apibrėžtą veiksmų tėkmę galima taikyti kiekviename simuliacijos žingsnyje.



1 pav.: Komponento veiksmų apdorojant užklausą veiklos diagrama.

Būtina pateikti nurodymus, kokius veiksmus atlieka serveris. Pateikiama esminių serverio veiksmų diagrama:



2 pav.: Serverio veiksmų apdorojant užduotis veiklos diagrama.

6. METODO TAIKYMO GAIRĖS

6.1. Duomenų, apibūdinančių modelio elgseną simuliacijos metu, rinkimo gairės

Siekiant turėti platesnes galimybes kaupti duomenis modelio simuliacijos vykdymo metu, buvo sukurtas agento tipas, pavadinimu „Stebėtojas“, kuris turi gebėjimą rinkti informaciją kiekvieno simuliacijos žingsnio metu ir ją kaupti. Tai gali būti informacija susijusi su modelio esybėmis: komponentais, serveriais, užklausomis, užduotimis bei jų elgsena ar būsenomis. Šio agento tipo agentai – infrastruktūros dalis, jie nedaro jokios įtakos pačiai simuliacijai. Duomenys kuriuos siūlo kaupti darbo autorius simuliacijos metu:

- Komponento užimtų gijų skaičius procentais, kiekvienu laiko (simuliacijos žingsnio) momentu;
- Nepavykusių užklausų visoje sistemoje skaičius procentais pagal priežastį, kiekvienu laiko momentu (vardiklis – bendras nepavykusių užklausų skaičius);
- Nepavykusių užklausų visoje sistemoje skaičius procentais, kiekvienu laiko momentu (vardiklis – visų užklausų skaičius);
- Kiekvienos užklausos bendra trukmė;
- Komponento siunčiamų įvykdytų užklausų trukmės vidurkis laiko momentu;
- Komponento siunčiamų neįvykdytų užklausų trukmės vidurkis laiko momentu;
- Komponento siunčiamų užklausų, kurios neatlieka jokių veiksmų apdorojimo metu, trukmės vidurkis laiko momentu;
- Komponento siunčiamų užklausų nusiuntimo trukmės vidurkis laiko momentu;
- Komponento siunčiamų užklausų atsako gavimo trukmės vidurkis laiko momentu;
- Nepavykusi užklausų, kurios neatlieka jokių veiksmų apdorojimo metu, visoje sistemoje skaičius procentais, laiko momentu;
- Užklausų, kurių vaikinės užklausos bendrina būseną, vidutinis vėlavimas, kuris skaičiuojamas iš vaikinių užklausų gavimo laiko (tas taškas laike, kuomet užklausa pasiekia komponentą) atimant tėvinės užklausos gavimo laiką;
- Komponentų, kurie bendrina būsenas, būsenų bendrinimo vėlavimas;
- Nepavykusių naudotojų siunčiamų užklausų skaičius procentais laiko momentu.

6.2. Architektūrinio sprendimo modeliavimas

Norint pagerinti metodo praktinį panaudojamumą, pravartu ją įgyvendinant numatyti galimybę greitai ir nesudėtingai sumodeliuoti reikalingą architektūrinį sprendimą. Yra keletas galimų būdų, kurie leistų tai pasiekti:

- Grafinės aplinkos sukūrimas, kuri leistų naudotojui kurti naudotojus, serverius, komponentus bei ryšius tarp jų nurodant šių elementų parametrus, jei šie neatitinka numatytų standartinių reikšmių. Vizualiai tai galėtų atrodyti kaip dauguma darbui su UML diagramomis skirtų įrankių. Taip pat, būtų galima po simuliacijos galima pateikti surinktus duomenis bei statistiką ir ją pavaizduoti vizualiai – pvz. grafikais;
- Parengti numatytų sprendimų šablonus (kurie galėtų atspindėti tam tikrus architektūrinius stilius ar keletą jų) aprašytus pasirinkta programavimo kalba, kuriuos būtų galima koreguoti pagal poreikį, taip išvengiant viso sprendimo kūrimo nuo nulio;
- Sukurti architektūrinio sprendimo generatorių, kuriam nustačius tokius parametrus kaip: naudotojų skaičių, komponentų skaičių, ryšių tarp komponentų skaičių, vidutinį vienam serveriui tenkančių komponentų skaičių, ryšių tipą ir kitus, jis gebėtų automatiškai sugeneruoti sprendimą pagal pateiktus parametrus.

Šiuos būdus galima naudoti tiek visus kartu, tiek kiekvieną atskirai, priklausomai nuo poreikių bei metodo naudotojų įgūdžių. Naudotojas, kuris geba programuoti, nesudėtingai susikurs sau reikiamą sprendimą kodo pagalba, o nemokančiam patogiausia tai būtų atlikti naudojant grafinę aplinką.

7. ĮRANKIO LEIDŽIANČIO MOEDELIUOTI BEI VYKDYTI SIMULIACIJĄ KŪRIMAS

Siekiant turėti galimybę vertinti savybes, kurias suteikia architektūriniai sprendimai bei stiliai, tinkle išskirstytoms programų sistemoms, reikia pagal ankstesniame skyriuje (5 skyrius) išdėstytus metodo nurodymus sukurti įrankį, kuriuo būtų galima vykdyti modelio simuliaciją. Simuliacijos įrankio kūrimui buvo pasitelktas „Repast Simphony“ įrankis, kuris leidžia kurti ir esant poreikiui vizualizuoti agentinę simuliaciją. Simuliacija atliekama žingsniais. Vienas žingsnis – vienas laiko vienetas.

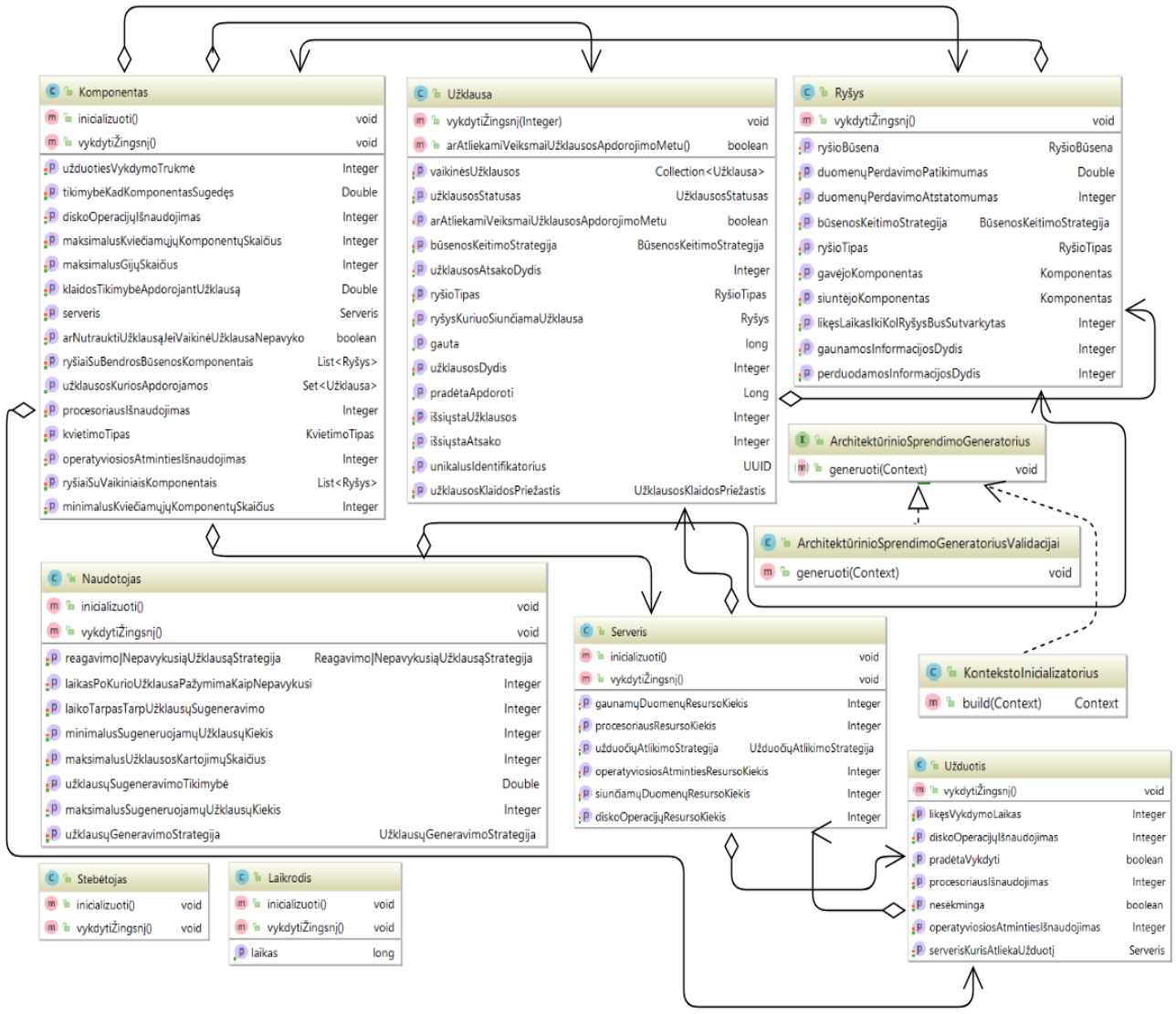
7.1. Simuliacijos eiga

Simuliacija susidaro iš trijų pagrindinių žingsnių:

- Inicializacijos;
- Vykdymo;
- Rezultatų išvedimo.

Inicializacijos metu yra sukuriamas kontekstas: sukuriami reikalingi agentai (tam tikras kiekis naudotojų, serverių, komponentų), nustatomos pradinės šių agentų atributų reikšmės. Vykdymo metu, periodiškai, vienodą kiekį kartų yra kviečiami pagrindiniai agentų metodai „vykdytiŽingsnį“. Jei inicializuojant simuliaciją buvo sukurtas agentas „Stebėtojas“, tai jo metodą „vykdytiŽingsnį“ kviečia paskutinį iš visų agentų, kad „Stebėtojas“ galėtų užfiksuoti visus per laiko vienetą atsitikusius įvykius, atsiradusius pokyčius. Paskutiniame simuliacijos etape yra išvedami duomenys, kuriuos surinko agentas „Stebėtojas“. Tuos duomenis galima papildyti išvestiniais duomenimis pagal poreikį (pasinaudoti statistiniais metodais ir paskaičiuoti pvz. standartinį nuokrypį ar pan.).

7.2. Įrankio struktūra



3 pav.: Modelio ir simuliacijos įgyvendinimo klasių diagrama su esminiais ryšiais.

Įrankis, leidžiantis kurti modelius bei vykdyti simuliacijas įgyvendintas naudojant objektinę paradigmą, JAVA programavimo kalbos 8 versiją. Tačiau metodas to primygtinai nereikalauja, toks sprendimas pasirinktas tik dėl patogumo bei autoriaus turimų įgūdžių. Tokį patį įrankį būtų galima įgyvendinti bet kuria programavimo kalba ir paradigma, tai neturėtų įtakos gaunamiems rezultatams.

Aukščiau pateiktoje klasių diagramoje dėl labai didelio kiekio ryšių nepateikti enumeratoriai (angl. *enumerator*). Iš viso jų yra 9. Toks kiekis enumeratorių nėra būtinas, tačiau darbo autorius dėl aiškumo vietoje loginio (angl. *bool*) tipo naudojo enumeratorius. Enumeratoriai bei jų atitikimas metodui:

- BūsenosKeitimoStrategija. Atitinka atributą R_{sc};
- KvietimoTipas. Atitinka atributą K_k;
- ReagavimoĮNepavykusiąUžklausąStrategija. Atitinka atributą V_k;
- RyšioBūsena. Atitinka atributą R_b;
- RyšioTipas. Atitinka atributą R_t;
- UžduočiųAtlikimoStrategija. Atitinka atributą S_s;
- UžklausosKlaidosPriežastis. Enumeratorius skirtas nustatyti, dėl kokios priežasties užklausa nepavyko. Padeda rinkti tikslesnę statistiką;
- UžklausosStatusas. 5.2 skyriuje aprašytas užklausai būdingas statusas;
- UžklausųGeneravimoStrategija. Atitinka atributą V_s.

8. METODO VALIDAVIMAS

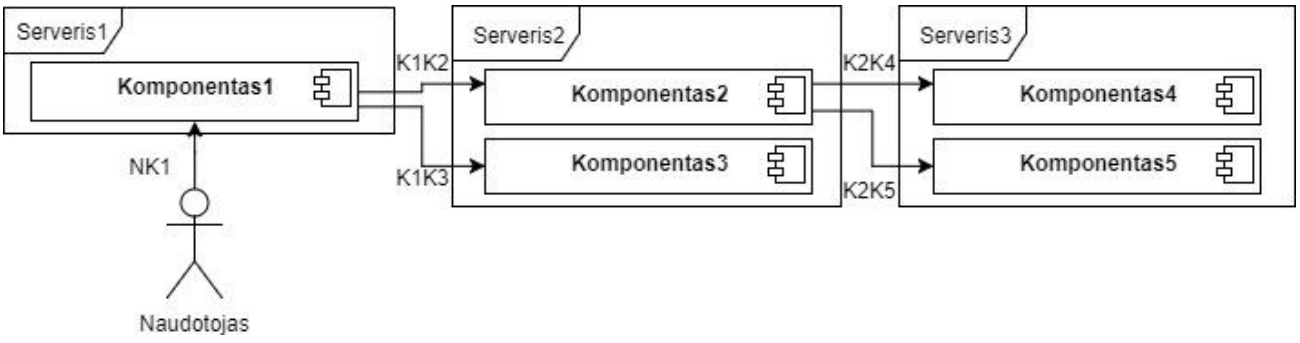
Siekiant įsitikinti metode atliktų pakeitimų, lyginant su Siliūno pateikta metodika, teisingumu, būtina jį validuoti. Metodo validacija atlikta tokia veiksmų seka:

- Sukurtas architektūrinio sprendimo generatorius, kuris metodo pritaikymo įrankyje leidžia sukurti modelį, panaudojant didžiąją dalį metodo galimybių;
- Įgyvendinta reali sistema, kuri atitinka ankstesniame žingsnyje sukurtą modelį;
- Pasirinkti metodo aspektai, kuriuos norima validuoti;
- Atliekami bandymai tiek su realia sistema, tiek vykdoma simuliacija metodo pritaikymo įrankyje;
- Surenkama statistika;
- Lyginami surinkti duomenys iš realios sistemos su duomenimis gautais simuliacijos metu.

Kiekvienas iš šių žingsnių bus detaliai aptartas sekančiuose šio skyriaus poskyriuose. Taip pat, bus pateikta ir šiek tiek kitokio pobūdžio validacija, kurios metu gauti matavimų rezultatai yra lyginami ne su realioje sistemoje surinktais matavimų rezultatais, o su kituose darbuose pateiktais rezultatais, taip užtikrinant, kad nėra didelių nuokrypių.

8.1. Architektūrinio sprendimo modelis validacijai

Validacijos pradžioje buvo sukurtas architektūrinis sprendimas, kuris bus naudojamas validuojant metodą. Jis susideda iš 5 komponentų, 1 naudotojo bei 3 serverių.



4 pav.: Modelio validacijai schema.

Toliau bus pateikti pradinės šio modelio agentų atributų reikšmės:

- **Serveris1**. S_p= 5000; S_a= 5000; S_d= 5000; S_s= EILĖ; S_{ta}= 7000; S_{ti}= 7000;
- **Serveris2**. S_p= 5000; S_a= 5000; S_d= 5000; S_s= EILĖ; S_{ta}= 6000; S_{ti}= 6000;

- **Serveris3**. S_p= 5000; S_a= 5000; S_d= 5000; S_s= EILĖ; S_{ta}= 5000; S_{ti}= 5000;
- **Komponentas1**. K_k= VISUS; K_l= 200; K_e= 0,01; K_{dw}= 0,01; K_r= [K1K2, K1K3]; K_s= [Serveris1]; K_{cpu}= 100;K_o= 100; K_d= 100; K_t= 10; K_{sh}= []; K_{nvn}= TAIP;
- **Komponentas2**. K_k= VISUS; K_l= 200; K_e= 0,001; K_{dw}= 0,01; K_r= []; K_s= [Serveris2]; K_{cpu}= 100; K_o= 100; K_d= 100; K_t= 20; K_{sh}= [K2K4, K2K5]; K_{nvn}= TAIP;
- **Komponentas3**. K_k= VISUS; K_l= 200; K_e= 0,001; K_{dw}= 0,01; K_r= []; K_s= [Serveris2]; K_{cpu}= 100; K_o= 100; K_d= 100; K_t= 30; K_{sh}= []; K_{nvn}= TAIP;
- **Komponentas4**. K_k= VISUS; K_l= 200; K_e= 0,01; K_{dw}= 0,01; K_r= []; K_s= [Serveris1]; K_{cpu}= 100; K_o= 100; K_d= 100; K_t= 5; K_{sh}= []; K_{nvn}= TAIP;
- **Komponentas5**. K_k= VISUS; K_l= 200; K_e= 0,01; K_{dw}= 0,01; K_r= []; K_s= [Serveris1]; K_{cpu}= 100; K_o= 100; K_d= 100; K_t= 5; K_{sh}= []; K_{nvn}= TAIP;
- **Ryšys NK1**. R_{si}= 150; R_{gi}= 100; R_t= SINCHRONINIS; R_{gk}= Komponentas1; R_{sk}= -; R_{dp}= 0,01; R_{dpa}= 1; R_{sc}= KEIČIA;
- **Ryšys K1K2**. R_{si}= 500; R_{gi}= 300; R_t= SINCHRONINIS; R_{gk}= Komponentas2; R_{sk}= Komponentas1; R_{dp}= 0,01; R_{dpa}= 1; R_{sc}= KEIČIA;
- **Ryšys K1K3**. R_{si}= 700; R_{gi}= 400; R_t= ASINCHRONINIS; R_{gk}= Komponentas3; R_{sk}= Komponentas1; R_{dp}= 0,01; R_{dpa}= 1; R_{sc}= NEKEIČIA;
- **Ryšys K2K4**. R_{si}= 700; R_{gi}= 1; R_t= SINCHRONINIS; R_{gk}= Komponentas4; R_{sk}= Komponentas2; R_{dp}= 0,01; R_{dpa}= 1; R_{sc}= KEIČIA;
- **Ryšys K2K5**. R_{si}= 350; R_{gi}= 1; R_t= SINCHRONINIS; R_{gk}= Komponentas5; R_{sk}= Komponentas2; R_{dp}= 0,01; R_{dpa}= 1; R_{sc}= KEIČIA;
- **Naudotojas**. V_r= NK1; V_s= REGULIARI; V_t= 1; V_{umax}= 1; V_{umin}= 1; V_k= NEKARTOTI; V_{ul}= 200;

8.2. Reali sistema atitinkanti modelį

Validacijai vykdyti buvo sukurta reali sistema, kuri susideda iš tokio pačio kiekio komponentų ir ryšių tarp jų bei naudotojo, kuris generuoja užklausas. Vietoje 3 atskirų serverių bus naudojami kiti sprendimai, kurie leistų atkurti komponentų veikimą juose.

Autorius pasirinko komponentus įgyvendinti kaip tinklo programas, kurios tarpusavyje komunikuoja HTTP protokolu. Šiam tikslui pasiekti buvo pasitelkti JAVA programavimo kalbos, Spring karkaso įrankiai: Spring Boot (aplikacijos paleidimui), Spring Web (komunikacijai HTTP

protokolu). Užklausoms siųsti naudojami Spring Web rinkinyje esantys įrankiai RestTemplate ir AsynRestTemplate – atitinkamai sinchroninėms ir asinchroninėms užklausoms siųsti. Užklausų bei atsakų į jas dydis nustatomas sukuriant simbolių eilutę tokio dydžio, kokio reikia ir ją atiduodant kaip užklausos atsaką. Užduočių vykdymas imituojamas kaip programos gijų sustabdymas užduoties įvykdymo laikui.

Naudotojas įgyvendinamas kaip programinė įranga skirta testuoti programų sistemų nefunkcinėms savybėms – Apache JMeter. Ši nemokama, atviro kodo programinė įranga leidžia generuoti užklausas bei konfigūruoti generavimą, rinkti statistiką, ją atvaizduoti grafiškai.

8.3. Aplinkos aprašas

Aplinka, kurioje atliekami bandymai, gali turėti nežymios įtakos gautiems absoliutiems renkamų duomenų įverčiams, todėl darbo autorius šiame skyriuje pateikia detalią informaciją apibūdinančią naudotą techninę bei programinę įrangą.

8.3.1. Naudota techninė įranga

- Procesorius: Intel Core i5-4210U, 2.7 Ghz, 2 branduoliai;
- Operatyvioji atmintis: 8.00 GB DDR3, dažnis 1600 Mhz.

8.3.2. Naudota programinė įranga

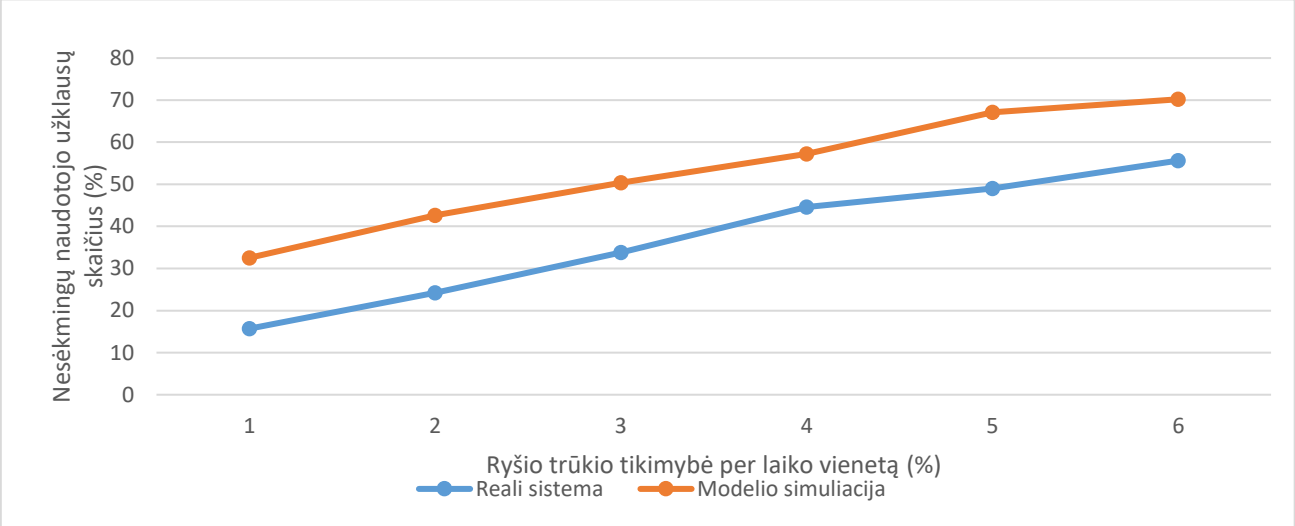
- Operacinė sistema: Windows 10 Pro, 64 bit;
- Programinės įrangos kūrimo įrankių rinkinys (angl. *software development kit*): Java SE Development Kit 1.8.0_73;
- Spartos ir kitų savybių matavimo įrankis: Apache JMeter 4.0;
- „Proxy“ serveris: Toxiproxy v2.1.2.

8.4. Ryšio patikimumo bei atstatomumo validacija

Validuojant ryšio patikimumą ir atstatomumą, visų pirma, reikia sukurti būdą, kaip realioje sistemoje valdomai atkurti minėtas ryšių savybes. T.y. tikimybę, kad ryšys einamuoju laiko momentu nutrūks ir bus nutrūkęs tam tikrą laiko kiekį. Sistemų architektai bei programuotojai, norėdami patikrinti, kaip veikia jų sukurtas produktas (tinklo aplikacijos) esant tinklo trūkiams neretai naudoja „proxy“ serverius, po kuriais paslepia savo aplikacijas, o tuomet atsiranda galimybė riboti tinklo srautą, padaryti aplikacijas nepasiekiamas ir t.t. Tokį būdą pasirinko ir šio darbo autorius. Buvo

pasirinktas atviro kodo „proxy“ serveris pavadinimu „Toxiproxy“ [Toxi18], kuris gali keisti bei valdyti TCP srautus. Kadangi komponentų ryšiai įgyvendinami HTTP protokolu, tai šis įrankis tiko, nes HTTP veikia TCP pagrindu. Buvo sukurta programa, kuri valdo „proxy“ serverį ir gali nustatyti, kurį komponentą padaryti nepasiekiamu (nes komponentai vienas su kitu bendrauja per „proxy“ serverį) ir kuriam laikui.

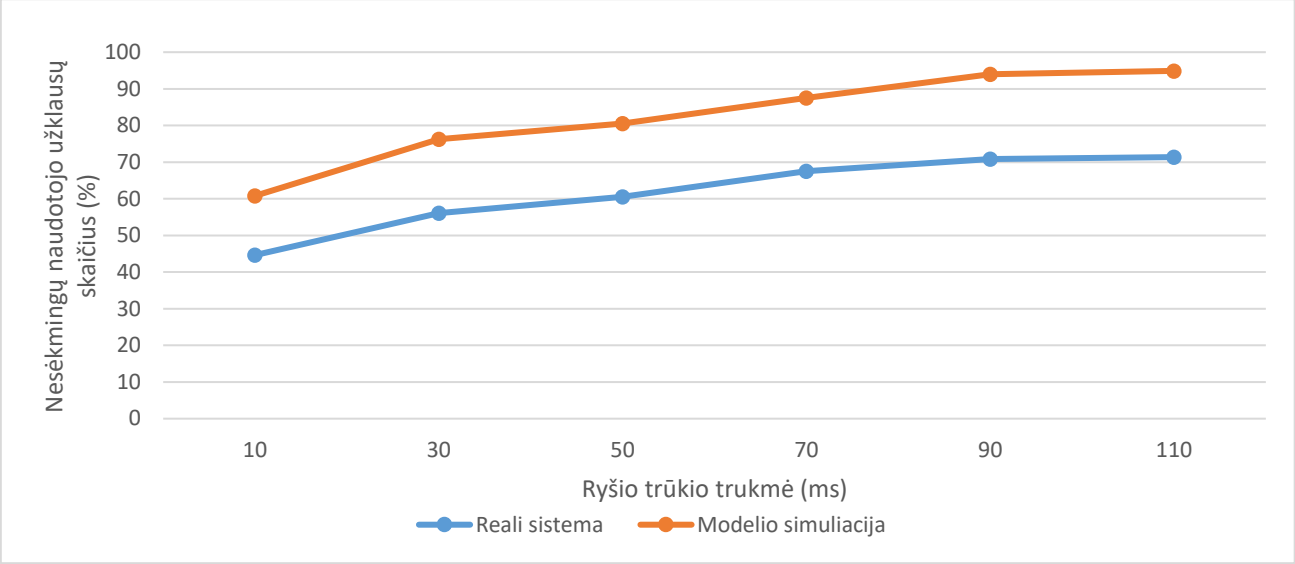
Pirmiausia buvo validuojamas ryšio atributas nurodantis tinklo trūkio tikimybę. Tai atliekama matuojant nesėkmingų užklausų, kurias siunčia naudotojas komponentui, kiekį, priklausomai nuo tinklo trūkio (ryšio patikimumo atributas) tikimybės. Matuojamos būtent naudotojo užklausų trukmės, nes naudotojui atsispindi visos sistemos veikimas: jei kuris nors komponentas veikia netinkamai (lėtai, su dideliu kiekiu klaidų ar pan.), naudotojas visuomet tai pajus, jei jo užklausų grandinėje tas komponentas yra. Keičiant visų sistemos ryšių atributo R_{dp} reikšmes nuo 1% iki 6% (nuo 0,01 iki 0.06), kiekvienu matavimu naudotojui sugeneruojant 2000 užklausų, buvo gauti matavimo rezultatai tiek simuliacijos pagalba, tiek realioje sistemoje. Jie atvaizduoti diagrama:



5 pav.: Nesėkmingų naudotojo užklausų skaičiaus procentais priklausomai nuo ryšio trūkio tikimybės per laiko vienetą diagrama.

Koreliacija tarp šių dviejų matavimų yra 0.99, kas reiškia, jog koreliacija labai stipri.

Toliau buvo validuojamas tinklo trūkio trukmės atributo modeliavimas metode. Tai atliekama matuojant nesėkmingų užklausų, kurias siunčia naudotojas komponentui, kiekį, priklausomai nuo tinklo trūkio trukmės (arba kitaip – atstatomumo) atributo reikšmės. Nustačius reikšmę $R_{dp} = 0{,}04$ ir keičiant trūkio trukmę (R_{gi}) nuo 1 iki 11 vienetų, buvo gauti matavimo rezultatai tiek simuliacijos pagalba, tiek realioje sistemoje:



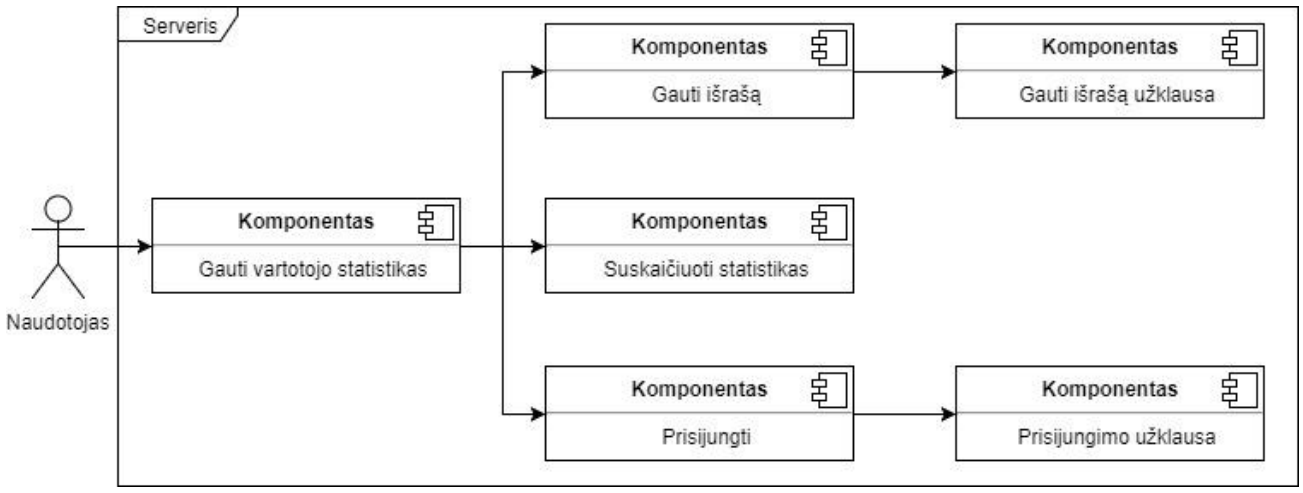
6 pav.: Nesėkmingų naudotojo užklausų skaičius procentais priklausomai nuo ryšio trūkio trukmės diagrama.

Koreliacija tarp šių dviejų dydžių – 0.997, kas rodo labai stiprią koreliaciją.

Palyginus to pačio modelio nesėkmingų užklausų skaičių procentais, patį modelį įgyvendinant remiantis metodu ir, taip pat, praktinėje programų sistemų inžinerijoje dažnai naudojamais įrankiais, galima teigti, kad ryšio patikimumo ir atstatomumo atributų reikšmių pokyčiai suteikia validžius pokyčius bendram klaidų skaičiui.

8.5. Validacija lyginant su esamų tyrimų rezultatais

Šiame darbe minėtuose magistriniuose Siliūno bei Mikoliūno darbuose taip pat buvo atliktos validacijos. Kai kurios iš jų – validuota lyginant su realų panaudos atvejį atitinkančiomis sistemomis. Šio darbo autorius nusprendė atkurti Mikoliūno darbe sukurtą validacijos modelį ir patikrinti, ar nėra esminių skirtumų tarp gautų rezultatų.



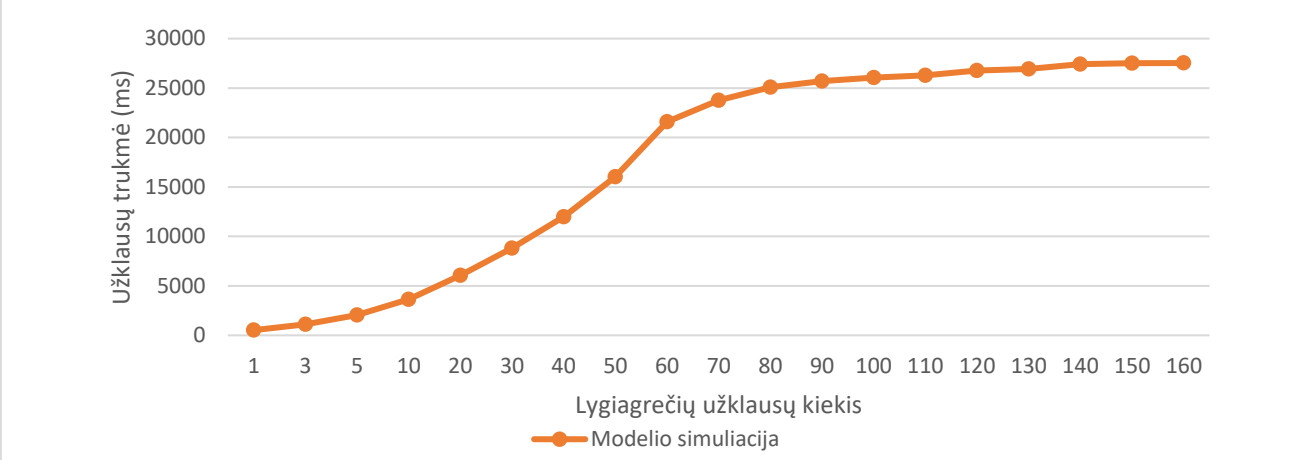
7 pav.: Mikoliūno darbe pateikiamo modelio skirto validacijai diagrama.

Privalu aprašyti atributų reikšmes, kurios buvo parinktos šiems komponentams. Visiems komponentams nustatytas maksimalus, 20 gijų, limitas. Serveris atlikinėja užklausas atsitiktine tvarka. Maksimali užklausos trukmė 30s. Nei vienas komponentas nenaudoja tinklo išteklių, nes modelyje atvaizduota sistema sudiegta į vieną serverį. Visi ryšiai yra sinchroniniai. Tikslesnės atributų reikšmės:

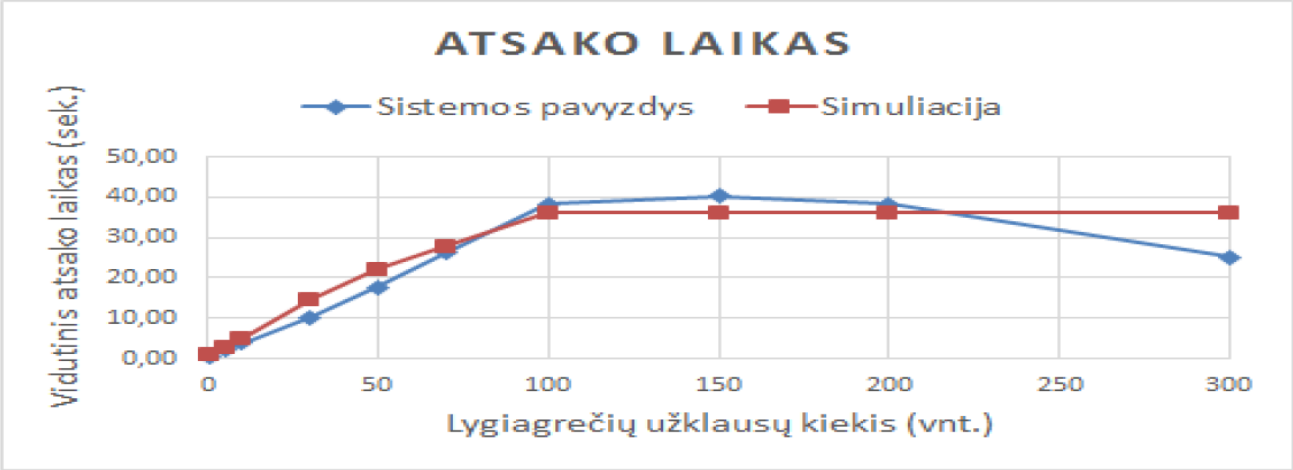
- „Gauti vartotojo statistikas“: sunaudojama po 5 proc. procesoriaus, operatyviosios atminties bei disko išteklių. Užduoties trukmė – 10ms;
- „Prisijungti“: sunaudojama po 5 proc. procesoriaus, operatyviosios atminties bei disko išteklių. Užduoties trukmė – 2ms;
- „Prisijungimo užklausa“: sunaudojama po 30 proc. procesoriaus bei disko išteklių, 5 proc. operatyviosios atminties išteklio. Užduoties trukmė – 100ms;
- „Gauti išrašą“ atributų reikšmės sutampa su „Gauti vartotojo statistikas“;
- „Gauti išrašą užklausa“: sunaudojama 60 proc. procesoriaus išteklių, 10 proc. operatyviosios atminties bei 90 proc. disko išteklių. Užduoties trukmė 350ms;
- „Suskaičiuoti statistikas“: sunaudojama 50 proc. procesoriaus išteklių, 15 proc. operatyviosios atminties bei 10 proc. disko išteklių. Užduoties trukmė 150ms.

Atkuriant tokį patį modelį naudojant šiame darbe aprašytą metodą, buvo „išjungtos“ galimos tinklo klaidos, tinklo pralaidumas nustatytas kaip artėjantis į begalybę, nes Mikoliūno metodikoje tai nebuvo numatyta.

Atkūrus modelį naudojantis šiame darbe aprašytu metodu buvo vykdoma simuliacija. Simuliacijos metu renkama naudotojo atliekamų užklausų vidutinės trukmės statistika pagal tai, kiek lygiagrečių užklausų naudotojas atlieka. Žemiau pateikiamos diagramos.



8 pav.: Simuliacijos rezultatai naudojantis šiame darbe pateiktu metodu.



9 pav.: Simuliacijos rezultatai Mikoliūno magistriniame darbe [Mik14].

Diagramose galima pastebėti tam tikrus paaiškinamus skirtumus. Simuliacijoje, kuri vykdyta remiantis šio darbo metodu, maksimali užklausos trukmė nesiekia 30s, kai tuo tarpu Mikoliūno darbe ji viršija 30s. Šis skirtumas paaiškinamas tuo, kad šiame darbe į statistikos skaičiavimus buvo įtraukiamos ir įvykdytos ir nesėkmingos užklausos. O nesėkmingų užklausų trukmė iš esmės yra trumpesnė, nei įvykdytų, nes užklausa gali būti pažymėta kaip nepavykusi kai tik pirmasis gavėjas ją ima vykdyti ir jam to padaryti nepavyksta. Kitas skirtumas yra toks, jog simuliacijoje, kuri įgyvendinta remiantis šio darbo metodu anksčiau pasiekiama maksimali užklausos trukmė. Mikoliūno darbe tai atsitinka prie 100 lygiagrečių užklausų, šiame darbe – prie ~85 lygiagrečių užklausų. Tai galima paaiškinti tuo, kad šio darbo metodas užklausos siuntimo laiką leidžia skaičiuoti pagal pralaidumą bei

užklausos dydį. Nors ir nustačius užklausos dydį į patį mažiausią įmanoma, o pralaidumą į patį didžiausią įmanomą, tačiau užklausos persiuntimui vis vien reikia vieno žingsnio, kuris turi įtakos.

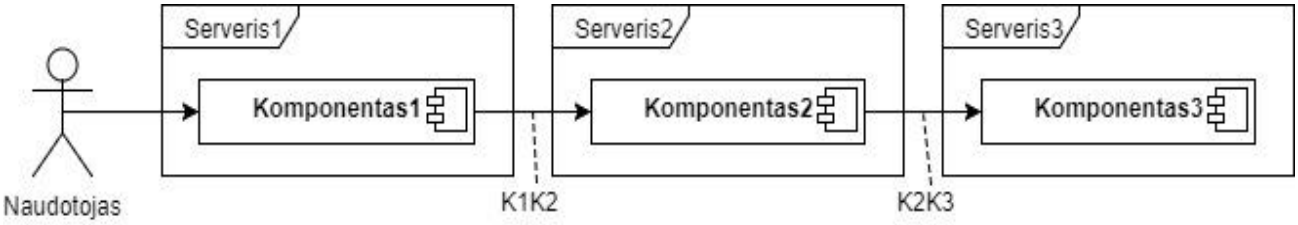
Apibendrinant galima pastebėti, kad skirtumai tarp simuliacijomis gautų rezultatų nėra dideli, o tai reiškia, jog šiame darbe aprašomas metodas yra sukurtas bei įgyvendintas be esminių, didelių klaidų.

9. METODO PRITAIKYMO PAVYZDŽIAI

Bet koks metodas, tam, kad turėtų vertę, turi būti panaudojamas. Siekiant įrodyti šiame darbe kuriamos metodo panaudojamumą, bus pateiktas pritaikymo pavyzdys. Kadangi šiame darbe metodas buvo papildytas galimybe modeliuoti tinklo trūkius, todėl pritaikymo pavyzdys bus glaudžiai su tuo susijęs.

9.1. Komponentų jungimo eiliškumo tyrimas

Neretai, praktinėje programų sistemų inžinerijoje modeliuojant sprendimus reikia keletą komponentų nuosekliai jungti į grandinę. Jungtys, arba ryšiai, gali būti skirtingo patikimumo. T.y. vienas ryšys gali dažniau trūkinėti ir ilgesniam laikui, nei kitas dėl pačių įvairiausių priežasčių. Šiame metodo pritaikymo pavyzdyje bus tiriama, kokią įtaką daro komponentų sujungimo eiliškumas. T.y. nustatinėjama, ar yra skirtumas, kaip sujungti ryšiais, kurie yra nevienodo patikimumo, komponentus. Ši situacija bus modeliuojama taip:



10 pav.: Metodo taikymo pavyzdžiui paruošto modelio diagrama.

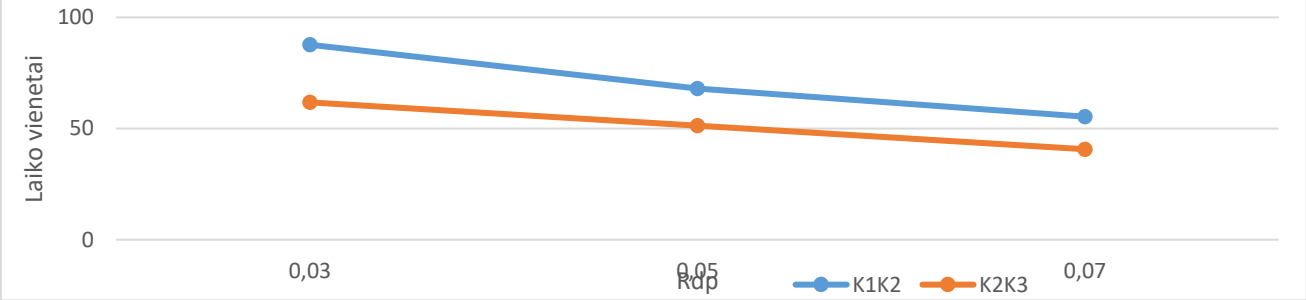
Visi komponentai bei serveriai yra vienodai sukonfigūruoti. Serveriai sukonfigūruoti taip, kad turėtų neribotus išteklius (išskyrus S_{ta}= 500 ir S_{ti}= 500) ir atlikinėtų užduotis atsitiktine tvarka. Komponentai turi po 200 gijų (atributas K_l) bei yra negendantys ir neklysta atlikinėdami užduotis. Užduoties vykdymo trukmė (atributas K_t) – 100 laiko vienetų. Naudotojas nuolat, kas 100 laiko vienetų, generuoja 50 lygiagrečių užklausų. Ryšiai K1K2 ir K2K3 yra vienodai patikimi: R_{dp} = 0,01 ir R_{dpa} = 5, o R_{si}= 20 bei R_{gi}= 20. Simuliacija nutraukiama, kai naudotojas būna išsiuntęs 15 tūkst. užklausų. Visi bandymai atliekami aplinkoje, kuri aprašyta šio darbo 8.3 skyriuje. Renkami tokie duomenys:

- Nesėkmingų naudotojo užklausų kiekis procentais;
- Sėkmingų naudotojo užklausų trukmė laiko vienetais;
- Nesėkmingų naudotojo užklausų trukmė laiko vienetais;
- Komponento užimtų gijų skaičius procentais.

Bandymai atlikti su tokiomis atributų reikšmėmis (nurodomi tik tie, kurie keičiami):

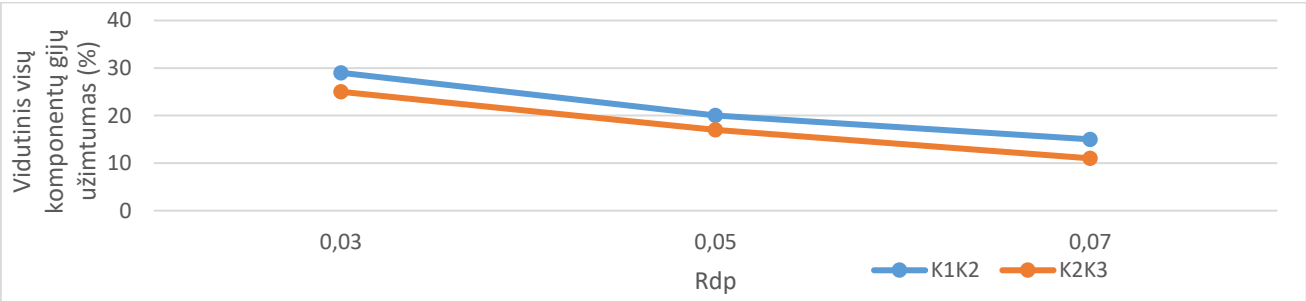
- Nieko nekeičiant.
- K1K2 ryšio R_{dp} = 0,03.
- K1K2 ryšio R_{dp} = 0,05.
- K1K2 ryšio R_{dp} = 0,07.
- K2K3 ryšio R_{dp} = 0,03.
- K2K3 ryšio R_{dp} = 0,05.
- K2K3 ryšio R_{dp} = 0,07.

Sėkmingų naudotojo užklausų trukmė keičiant aukščiau minėtas atributų reikšmes nesikeitė, išliko lygi ~345 laiko vienetams. Nesėkmingų naudotojo užklausų kiekis procentais keitėsi (didėjant R_{dp} didėjo), tačiau nepriklausomai nuo to, kuriam ryšiui buvo keičiama R_{dp} atributo reikšmė, pokytis buvo vienodas. Išsiskyrė tik užimtų gijų dalis procentais bei nepavykusių naudotojo užklausų trukmė.



11 pav.: Nepavykusių naudotojo užklausų trukmė priklausomai nuo Rdp atributo reikšmės.

Visų pirma, būtina paminėti, jog kuo užklausa greičiau pažymima, kaip nepavykusi, tuo visa sistema yra mažiau apkraunama. Ši teigiama sistemų savybė vadinama (angl. *fail-fast*). Kaip matome, padidinus „toliau“ nuo naudotojo esančio ryšio nepatikimumą, ši savybė yra išpildoma geriau. Tą patvirtina ir gijų užimtumo diagrama:



12 pav.: Vidutinis visų komponentų gijų užimtumas priklausomai nuo Rdp atributo reikšmės.

Matome, jog nepatikimas ryšys esantis „toliau“ nuo naudotojo grandinėje leidžia turėti daugiau laisvų gijų.

Atlikus tyrimą remiantis metodu galima teigti, jog komponentai, kurie yra pasiekiami nepatikimais ryšiais turėtų būti grandinės gale, jei norima apkrauti komponentus mažiau.

9.2. PACELC savybių tyrimas

Sekantis metodo pritaikymo pavyzdys susijęs su kompromisų, kurie apibūdinami PACELC teoremoje, tyrinėjimu. Pagrindinė šios bei CAP teoremos idėja yra ta, jog projektuojant tinkle išskirstytas sistemas tenka pasirinkti tarp pasiekiamumo, trumpo atsako laiko ir vientisumo. Šios architektūrinio sprendimo savybės negali būti įvertintos logine reikšme (t.y. „taip“ arba „ne“), nes pvz. sistema gali būti sudaryta iš didelio kiekio komponentų, todėl skirtingos sistemos dalys gali pasirinkti skirtingą elgsenos strategiją įvykus tinklo trūkiui. Taip pat, atsako laikas bei sistemos „nevientisumas“ (pvz. vienas iš galimų būdų įvertinti – vertinti laiku, kuomet sistema ar dalis sistemos būna „nevientisa“) gali būti labai skirtingi: kelių milisekundžių „nevientisumas“ nebūtų pastebėtas sistemos naudotojo ir atvirkščiai – kelių minučių „nevientisumas“ naudotoją gali klaidinti ir erzinti. Iš to galima daryti išvadą, jog svarbus ir kitas aspektas – kiek sistemos būsenos „nevientisumą“ pastebi naudotojas, nes galutinis bet kurios sistemos tikslas yra pasiekti, jog naudotojas būtų patenkintas.

Architektūriniai sprendimai pasirinkti tyrimui – grįsti SOA stiliumi. Darbo autorius norėdamas pagreitinti tokių sprendimų modeliavimą sukūrė generatorių, kuris sukuria agentinį sistemos modelį pagal pateiktus parametrus. Parametrai parinkti pagal tai, kokias sprendimo savybes bus norima tirti. Generatoriaus išeiga – metodo taikymo įrankyje sukuriamas objektinės paradigmos objektų rinkinys. Generatoriaus parametrai:

- G_{st} – skaičius servisų, kurie atsakingi už skirtingą sritį. Pvz. sprendimas gali susidaryti iš servisų, kurie atsakingi už naudotojų, dokumentų, failų ir pan. valdymą. Šis parametras nurodo, kiek tokių skirtingų sričių sprendime yra. Pati sritis, jos pobūdis, neturi jokios įtakos modelio generavimui ar simuliacijos rezultatams;
- G_r – skaičius komponentų, kiekvienos skirtingos srities servisui, su kuriais būsena yra bendrinama. Pvz. naudotojų valdymo servisas modeliuojamame sprendime gali turėti daugiau nei vieną kopiją, t.y. gali būti dublikuotas;
- G_b – būsenos bendrinimo tipas. „Šeimininkas/tarnas“ (angl. *master-slave*) ar „multi-šeimininkas“ (angl. *multi-master*). Pirmuoju atveju, būseną keičiančios užklausos siunčiamos

tik vienam iš pasirinktų komponentų, kuris įvardijamas kaip „šeimininkas“. Šio komponento užduotis – bendrinti užklausą su savo „tarnais“, kad esant situacijai, kai naudotojas pasikreipia į komponentą su tikslu gauti būseną, o ne ją keisti, „tarnas“ galėtų grąžinti kiek įmanoma naujesnę būseną, kurią yra gavęs iš „šeimininko“. „Šeimininkas“ gali priimti tiek užklausas, kurios keičia būsena, tiek tas, kurios nekeičia. Antrasis, „multi-šeimininko“ būsenos bendrinimo tipas, įgyvendinamas priešingai – kiekvienas iš komponentų gali priimti bet kokias užklausas, tiek tas, kurios keičia būseną, tiek tas, kurios nekeičia. Komponentas gavęs užklausą, kuri keičia būseną, išsiunčia ją kitiems komponentams, su kuriais tą būseną bendrina;

- G_a – asinchroninių ryšių, tarp skirtingų sričių servisų, kiekis procentais sprendimo modelyje;
- G_{nk} – naudotojų skaičius, kurie keičia būseną, vienos srities servisui;
- G_{nn} – naudotojų skaičius, kurie nekeičia būsenos, vienos srities servisui;
- G_j – skaičius kitos srities servisų, kurie yra kviečiami iš serviso;
- G_{rt} – būsenos bendrinimo ryšio tipas – sinchroninis arba asinchroninis. Jei užklausa, bendrinanti būseną, yra sinchroninė, tuomet į nepavykusią užklausą yra atsižvelgiama ir tėvinės užklausos yra pažymimos kaip nepavykusios. Jei asinchroninė – esant nesėkmingai būsenos bendrinimo užklausai, būsena esanti pas užklausos gavėją nepakinta, tačiau tėvinės užklausos gali būti sėkmingos. Tokiu būdu atsiranda tikimybė, jog naudotojas pastebės nevientisumą;
- G_t – laikas sekundėmis, kuris skiriamas kuriant architektūrinio sprendimo topologiją.

Taip pat, generatoriui būtina paduoti pavyzdines naudotojo kuris keičia būseną bei naudotojo kuris nekeičia būsenos, serverio, komponento, ryšio tarp komponentų, ryšio tarp naudotojo bei komponento konfigūracijas, kurias generatorius naudoja kuriant kiekvieną iš minėtų modelio elementų. Laikas, skiriamas topologijos sudarymui turi būti nurodytas, nes konstruojant ją, generuojamas atsitiktinis orientuotas aciklinis (aciklinis, kad nesusidarytų užklausų ciklai galintys iškreipti nustatinėjamas architektūrinio sprendimo savybes) grafas, siekiant, kad kiek įmanoma didesnis skaičius servisų kviestų generatoriaus parametruose nurodytą skaičių kitų sričių servisų. Algoritmas, kuris tai atlieka, veikia perrinkimo būdu ir gana greitai suranda patenkinamą topologiją, tačiau norint įsitikinti, kad geresnis (kuomet daugiau servisų kviečia nurodytąjį skaičių servisų, generatoriaus parametre) sprendimas neegzistuoja – trunka labai ilgai.

Scenarijai, kurie bus tiriami šiame metodo pritaikymo pavyzdyje:

- Ryšio (kuriuo yra bendrinama būsena) patikimumo ir atstatomumo įtaka vientisumui. T.y. kaip kinta dažnis įvykio, kuomet ne pati naujausia būsena yra grąžinama naudotojui, kuomet ryšio patikimumas, atstatomumas auga ar mažėja;
- Dublikatų skaičiaus įtaka užklausos trukmei. T.y. kaip kinta užklausos trukmė, didinant bei mažinant dublikatų skaičių;
- Dublikatų skaičiaus įtaka naudotojo pastebimam vientisumui. T.y. kaip kinta įvykių, kai ne naujausia būsena grąžinama naudotojui, dažnis, keičiant dublikatų skaičių;
- Būsenos bendrinimo tipo įtaka užklausos trukmei bei vientisumui. T.y. kaip pasikeičia užklausos trukmė bei naudotojo pastebimas vientisumas, kuomet pakeičiamas būsenos bendrinimo tipas.

9.2.1. Tyrimo objekto (architektūrinio sprendimo) aprašas

Prieš atliekant tyrimą, naudojantis ankstesniame skyriuje aprašytu generatoriumi, buvo sugeneruotas architektūrinis sprendimas. Žemiau pateikiami generatoriaus įeigos parametrai bei duomenys apie sugeneruotą sprendimą (modelį):

- G_{st} – 50; G_r – 2; G_b – „šeimininkas/tarnas“; G_a – 50 %; G_{nk} – 1; G_{nn} – 2; G_j – 5; G_{rt} – asinchroninis; G_t – 60s.

Parinktas optimalus kiekis servisų – 50, jog būtų galima analizuoti reiškinius, kurie kyla iš servisų tarpusavio sąveikos, taip pat, mažesnis kiekis servisų galėtų turėti įtakos rezultatų nestabilumui. Pusė iš visų ryšių tarp komponentų yra asinchroniniai, nes įprastai, kuo didesnis jų kiekis yra siekiamybė, tačiau architektūriniuose sprendimuose retai apsieinama be sinchroninių ryšių. Rezultate sugeneruojamas sprendimas, kuriame 24 iš 50 servisų kviečia po 5 atsitiktinius kitos srities servisus, o likę nekviečia kitų servisų. Parenkant pradinius generuojamo modelio parametrus, buvo stengtasi nesirinkti kraštutinių reikšmių, nepopuliarių sprendimų praktinėje sistemų inžinerijoje. Pvz. „šeiminkas/tarnas“ būsenos bendrinimo tipas yra gerokai dažniau sutinkamas bei lengviau įgyvendinamas nei minėta jo alternatyva. Taip pat, buvo stengtasi parinkti tokius komponentų bei serverių pajėgumą ir tokią naudotojų sukeliamą apkrovą, kad sumodeliuotas sprendimas simuliacijos metu nebūtų kritiškai apkrautas (kuomet didžioji dalis užklausų yra atmetamos dėl pasibaigusio vykdymo laiko), tačiau tuo pačiu, kad apkrova būtų juntama. Pavyzdinio serverio, kuris paduodamas į sprendimo generatorių parametrai:

- S_p= 1000; S_a= 1000; S_d= 1000; S_s= EILĖ; S_{ta}= 1000; S_{ti}= 1000.

Pavyzdinio naudotojo, kurio užklausos keičia būseną parametrai:

- V_s= REGULIARI; V_t= 1; V_{umax}= 2; V_{umin}= 1; V_k= NEKARTOTI; V_{ul}= 10.

Pavyzdinio naudotojo, kurio užklausos nekeičia būsenos parametrai:

- V_s= REGULIARI; V_t= 1; V_{umax}= 5; V_{umin}= 1; V_k= NEKARTOTI; V_{ul}= 10.

Pavyzdinio ryšio tarp komponentų parametrai:

- R_{si}=100; R_{gi}=100; R_{dp}= 0,01; R_{dpa}= 1.

Pavyzdinio ryšio tarp naudotojo bei komponento parametrai:

- R_{si}=100; R_{gi}=100; R_t= SINCHRONINIS; R_{dp}= 0; R_{dpa}= 0.

Pavyzdinio komponento parametrai:

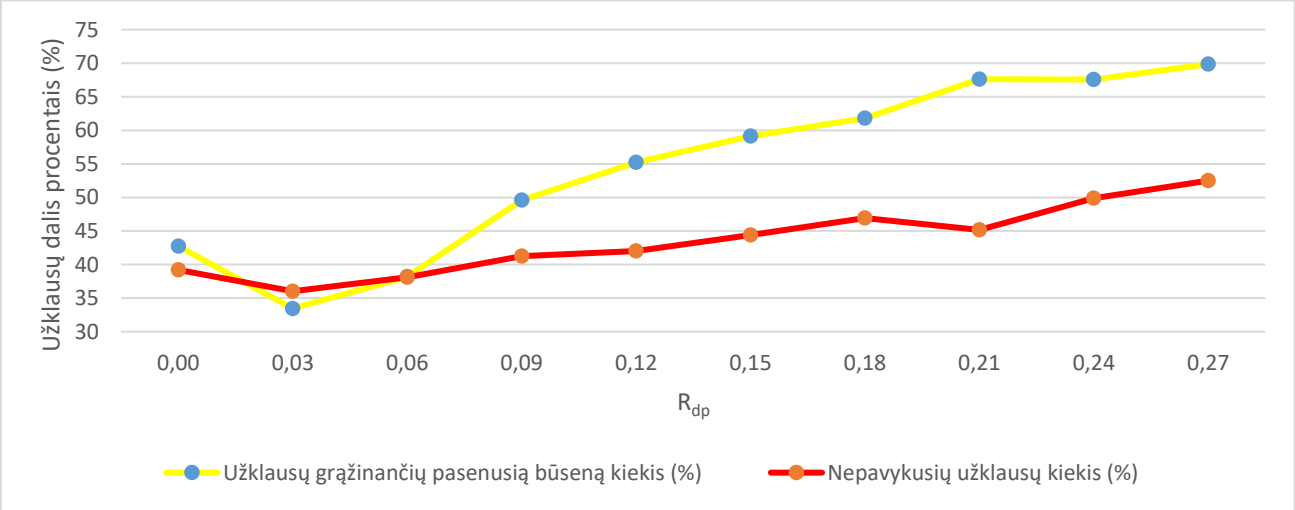
- K_k= VISUS; K_l= 100; K_e= 0,01; K_{dw}= 0,01; K_{cpu}= 100; K_o= 100; K_d= 100; K_t= 5; K_{nvn}= TAIP.

Sugeneruotas sprendimas naudojamas simuliacijoje, kuri stabdoma po 5000 žingsnių. Sustabdžius įvykdomi reikalingi skaičiavimai surinkti statistikai.

9.2.2. Ryšio patikimumo ir atstatomumo įtakos vientisumui tyrimas

Buvo atlikti bandymai, siekiant ištirti ryšio patikimumo ir atstatomumo įtaką vientisumui. Vientisumas šiame tyrime vertinamas per procentinį sėkmingų, nekeičiančių būsenos, užklausų kiekį, kurios grąžina ne pačią naujausią komponento būseną. Skaičiuoti šią reikšmę – agento „Stebėtojas“ atsakomybė. Tai atliekama nustatant kokią būseną kiekvienas iš servisų turėtų grąžinti kiekvienu laiko momentu ir lyginant ją su serviso komponentų užklausose grąžinamomis būsenomis. Nustatymas, kokią būseną turėtų grąžinti servisas einamuoju laiko momentu atliekamas nesudėtingai: paimama paskutinė užklausa, kuri keičia būseną ir kuri buvo pradėta apdoroti viename iš serviso komponentų. Iš jos paimama jos siųsta būsena, kuri ir yra ta būsena, kurią tuo laiko momentu turėtų grąžinti visi serviso komponentai.

Pirmasis bandymas atliktas siekiant nustatyti, kokią įtaką ryšio patikimumas turi vientisumui. Buvo keičiamas tik sprendimo generatoriaus, pavyzdinio ryšio tarp komponentų parametras R_{dp}. Gauti rezultatai pateikiami diagramoje.

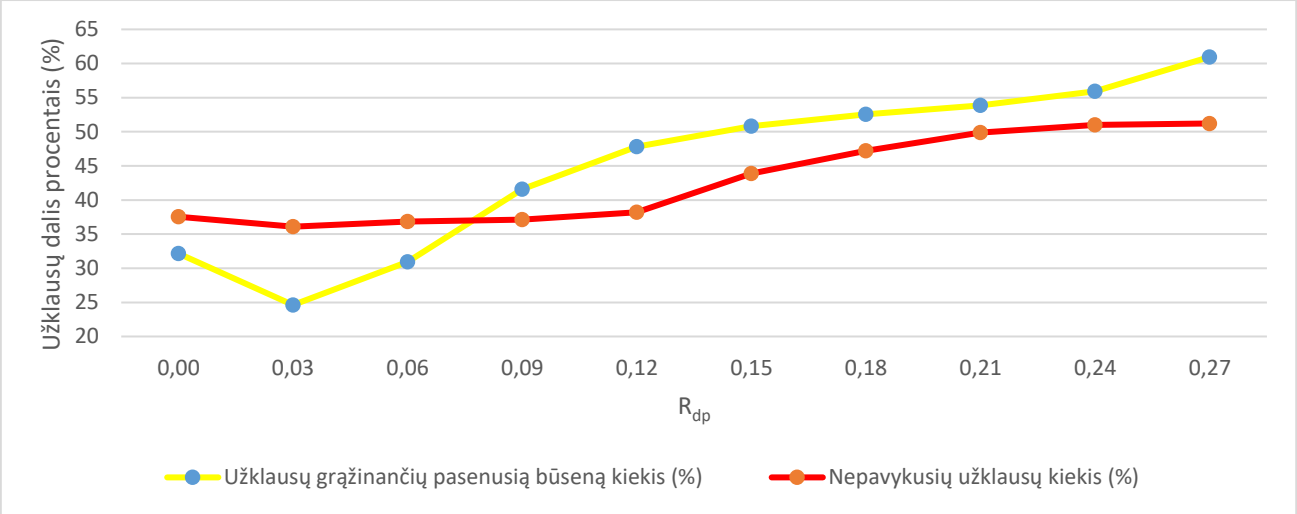


13 pav.: Užklausų grąžinančių pasenusią būseną kiekio priklausomybė nuo ryšio patikimumo kartu su nepavykusių užklausų kiekiu, kai naudotojo, kurio užklausos nekeičia būsenos parametras $V_{umax} = 5$.

Visų pirma pastebimas tiek nepavykusių, tiek grąžinančių seną būseną užklausų kiekio kritimas, kuomet $R_{dp} = 0,03$. Taip pat, pastebimas ir gana didelis (nors tai yra minimali reikšmė) užklausų grąžinančių pasenusią būseną kiekis tame pačiame $R_{dp} = 0,03$ taške – 33.46 %. Iš diagramos matyti,

kad egzistuoja stipri koreliacija (0.94) tarp klaidingų užklausų ir užklausų grąžinančių pasenusią būseną.

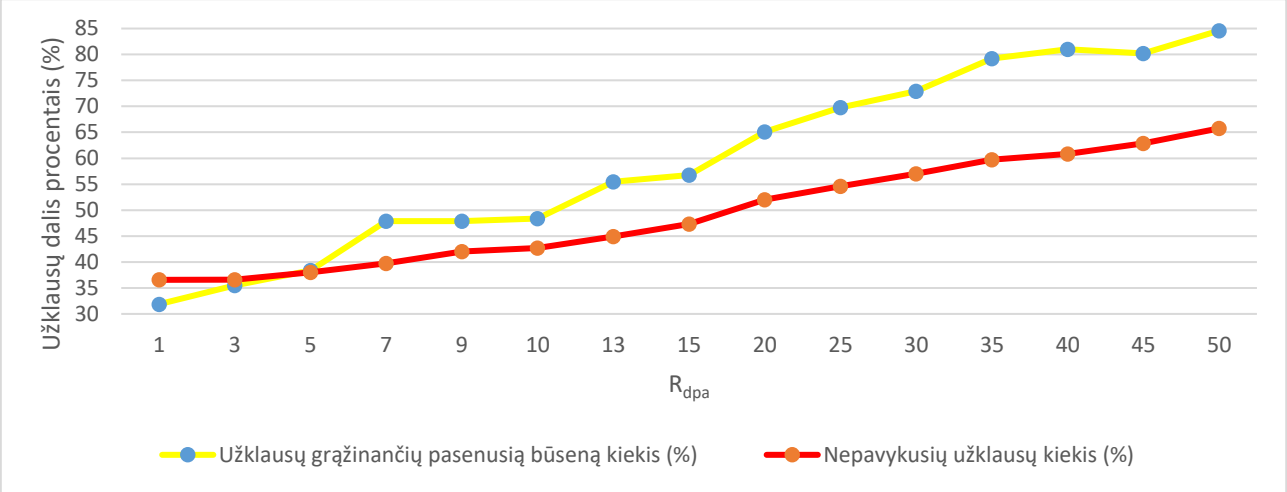
Antrasis bandymas atliktas iškėlus hipotezę, jog sumažėjus užklausų kiekiui t.y. sistemos apkrovai, užklausų, grąžinančių pasenusią būseną, procentinis kiekis turėtų mažėti. Todėl naudotojo, kurio užklausos nekeičia būsenos parametras V_{umax} pakeistas į 2 ir prieš tai atliktas bandymas pakartotas. Gauti rezultatai:



14 pav.: Užklausų grąžinančių pasenusią būseną kiekio priklausomybė nuo ryšio patikimumo kartu su nepavykusių užklausų kiekiu, kai naudotojo, kurio užklausos nekeičia būsenos parametras $V_{umax} = 2$.

Pakeitus V_{umax} parametrą iš 5 į 2, naudotojų generuojamų užklausų, kurios nekeičia būsenos, kiekis sumažėjo 2 kartus, tačiau užklausų grąžinančių pasenusią būseną kiekis sumažėjo ne taip žymiai. Užklausų grąžinančių pasenusią būseną kiekio skirtumų tarp pirmojo ir antrojo bandymo vidurkis – 9.31%. Nepavykusių užklausų kiekio skirtumų vidurkis – tik 1.87 %, todėl galima teigti, jog klaidų skaičius beveik nepakito.

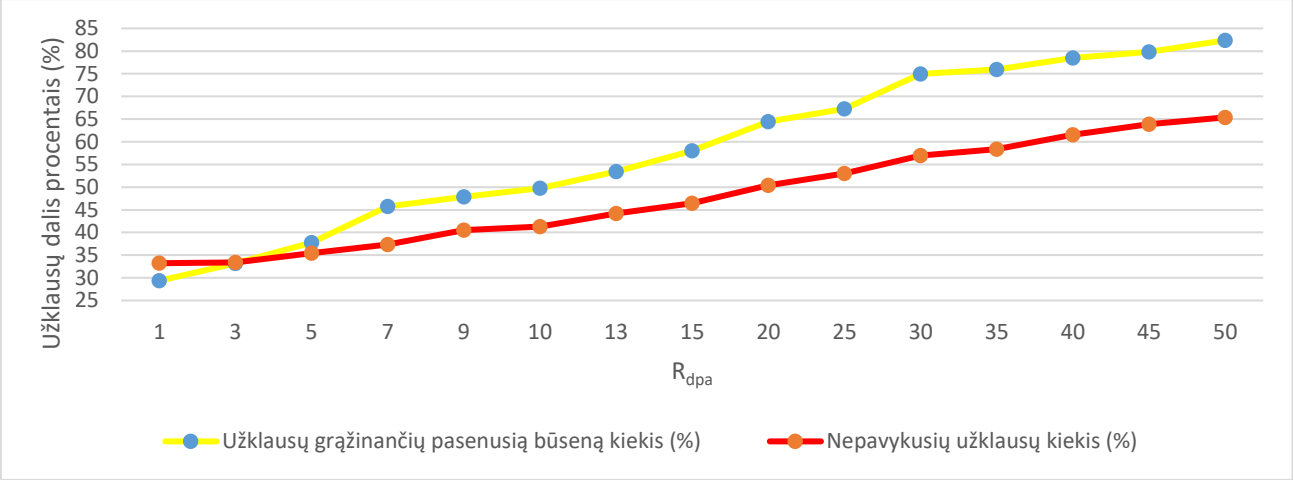
Trečiajame bandyme buvo tiriama, kokią įtaką turi ryšio atstatomumas vientisumui. Pavyzdinio ryšio tarp komponentų parametras R_{dp} pakeistas į 0,03, o pavyzdinio naudotojo, kurio užklausos nekeičia būsenos parametras $V_{umax} = 5$. Gauti rezultatai atvaizduoti diagramoje:



15 pav.: Užklausų grąžinančių pasenusią būseną kiekio priklausomybė nuo ryšio atstatomumo.

Pastebima, jog klaidingų ir pasenusią būseną grąžinančių būsenų kiekis stipriai koreliuoja (0.98). Akivaizdu ir tai, jog užklausų grąžinančių pasenusią būseną procentinis kiekis auga sparčiau, nei klaidingų užklausų kiekis.

Ketvirtajame bandyme vėl bandoma sumažinti naudotojų užklausų kiekį 2 kartus (naudotojo, kurio užklausos nekeičia būsenos parametras $V_{umax} = 2$) ir patikrinti, kaip keičiasi vientisumas, kuomet keičiamas ryšio atstatomumo parametras. Gauta diagrama:



16 pav.: Užklausų grąžinančių pasenusią būseną kiekio priklausomybė nuo ryšio atstatomumo, kai naudotojo, kurio užklausos nekeičia būsenos parametras $V_{umax} = 2$.

Užklausų grąžinančių pasenusią būseną kiekio skirtumų tarp trečiojo ir ketvirtojo bandymo vidurkis – 1.71%. Nepavykusių užklausų kiekio skirtumų tarp trečiojo ir ketvirtojo bandymo vidurkis – 1.51 %.

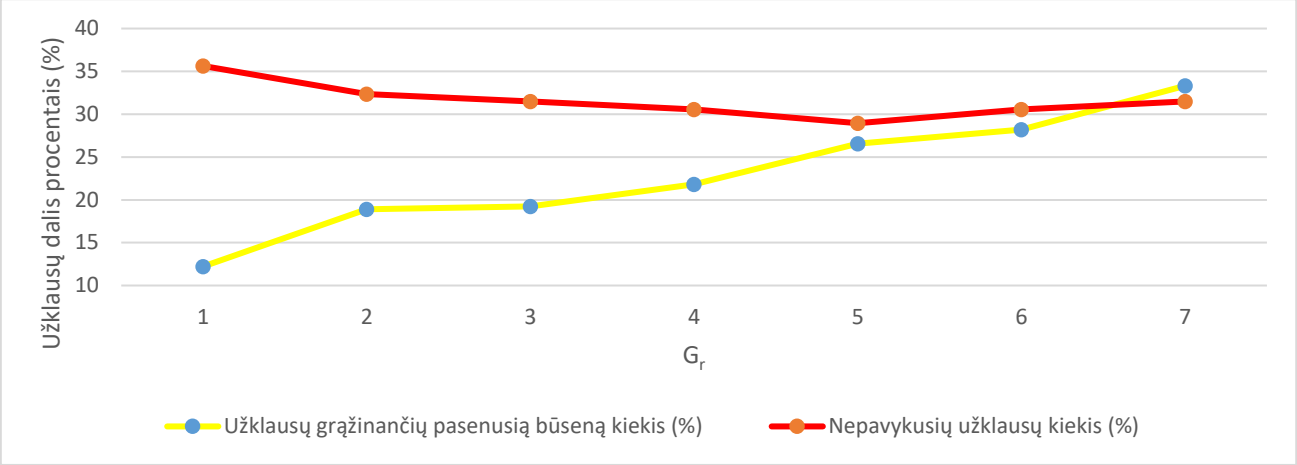
Analizuojant gautus rezultatus visų trijų bandymų metu, buvo pastebėti tam tikri dėsningumai. Bandymų, kurių metu buvo nustatinėjama ryšio patikimumo įtaka vientisumui, galima pastebėti, kad R_{dp} reikšmei esant intervale [0,03; 0,09] vientisumas mažėja sparčiausiai, o tai nusako jautrumą nedideliems tinklo trūkių kiekio pasikeitimams, kuomet tinklas yra gana patikimas. R_{dp} reikšmei esant intervale [0,00; 0,03] pasireiškia „saugiklio“ efektas: atsiradus tinklo trūkiams, sumažėja apkrova komponentams, nes sumažėja ir užklausų kiekis pasiekiantis komponentus. Sumažėjus apkrovai, mažesnis kiekis užklausų pažymimos kaip nepavykusios dėl pasibaigusio laiko, taip pat, sumažėjus apkrovai būsenos yra bendrinamos per trumpesnį laiką, todėl rečiau grąžinama pasenusi būsena. Tačiau antrajame bandyme sumažinus apkrovą du kartus, nesulaukta didelio pagerėjimo vientisumo užtikrinime, kas reiškia, jog jei sistema nėra perkrauta, užklausų skaičius neturi didelės įtakos vientisumui. Lyginant trečiąjį ir ketvirtąjį bandymus, pastebima, jog naudotojų generuojamas užklausų skaičius turi nykstamai mažą įtaką vientisumo pokyčiams. Vienintelis pastebimas skirtumas – tolydesnė ketvirtojo bandymo užklausų grąžinančių pasenusią būseną kiekio kreivė. Tai galima paaiškinti tuo, kad naudotojui generuojant nuo 1 užklausos iki 2 per vieną laiko vienetą susidaro mažesni siunčiamų, apdorojamų ir pan. užklausų pikai, nei tuo metu, kai naudotojas generuoja nuo 1 iki 5 užklausų.

9.2.3. Dublikatų skaičiaus įtakos užklausų trukmei bei vientisumui tyrimas

Šiame skyriuje bus tiriama serviso dublikatų skaičiaus įtaka vidutinei užklausų trukmei bei vidutiniam procentiniam kiekiui užklausų, kurios grąžina pasenusią būseną. Architektūrinio sprendimo generatoriui paduoti parametrai:

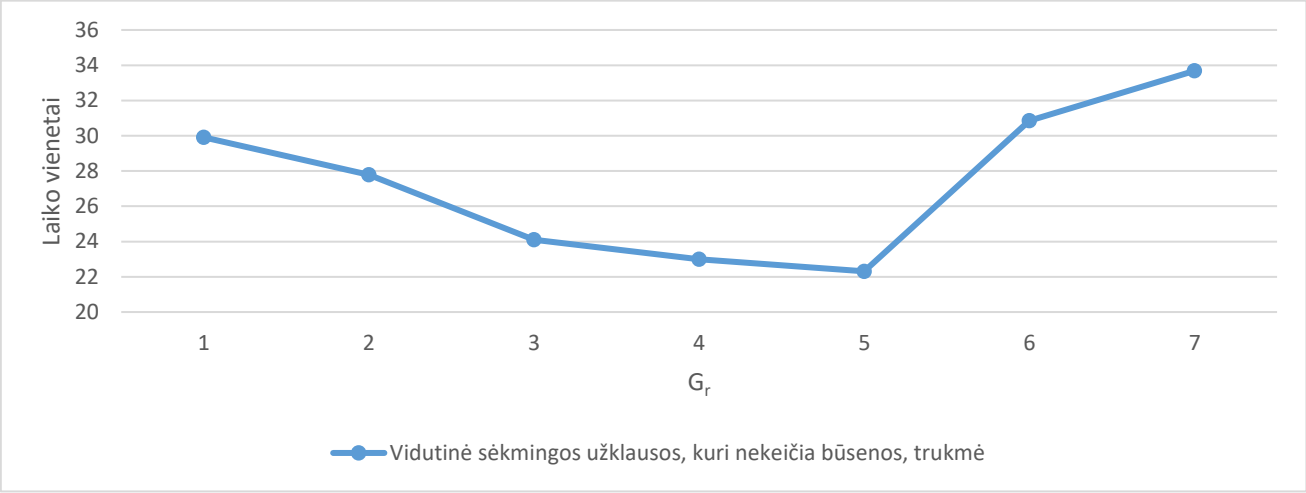
- Naudotojo, kurio užklausos nekeičia būsenos parametrai $V_{umax} = 7$ bei $V_{umin} = 5$;
- Pavyzdinio ryšio tarp komponentų parametrai $R_{dp} = 0,01$; $R_{dpa} = 1$.

Generatoriaus parametras G_r kinta nuo 1 iki 7 dublikatų vienam servisui. Gauti rezultatai:



17 pav.: Užklausų grąžinančių pasenusią būseną kiekio bei nepavykusių užklausų kiekio priklausomybė nuo dublikatų kiekio G_r.

Didėjant G_r pastebimas spartus užklausų grąžinančių pasenusią būseną kiekio didėjimas. Nepavykusių užklausų kiekis nežymiai, tačiau pradžioje (G_r iki 5) mažėja, o po to vėl ima augti.



18 pav.: Sėkmingos užklausos, kuri nekeičia būsenos, trukmės priklausomybė nuo dublikatų kiekio G_r.

Šioje diagramoje pastebimas užklausų trukmės kritimas iki $G_r = 5$. Nuo $G_r = 5$ užklausų trukmė ima sparčiai augti.

Pastebėtus pokyčius atliekant šį bandymą galima paaiškinti. Nepavykusių užklausų kiekis mažėja iki $G_r = 5$ dėl to, kad serviso patiriama apkrova pasiskirsto tarp dublikatų ir sumažėja užklausų, kurios pažymimos kaip klaidingos dėl pasibaigusio laiko kiekio. Tai įrodo ir 18 pav. grafikas, kuriame užklausos trukmė mažėja iki to pačio taško – $G_r = 5$. Augimas, kuris pastebimas nuo šio taško yra sukeltas padidėjusio kiekio užklausų (bei didesnės apkrovos „šeimininko“ komponentui), kurios bendrina būseną ir taip padidina servisų apkrovą, tuo pačiu padidindamas užklausų trukmę bei klaidų

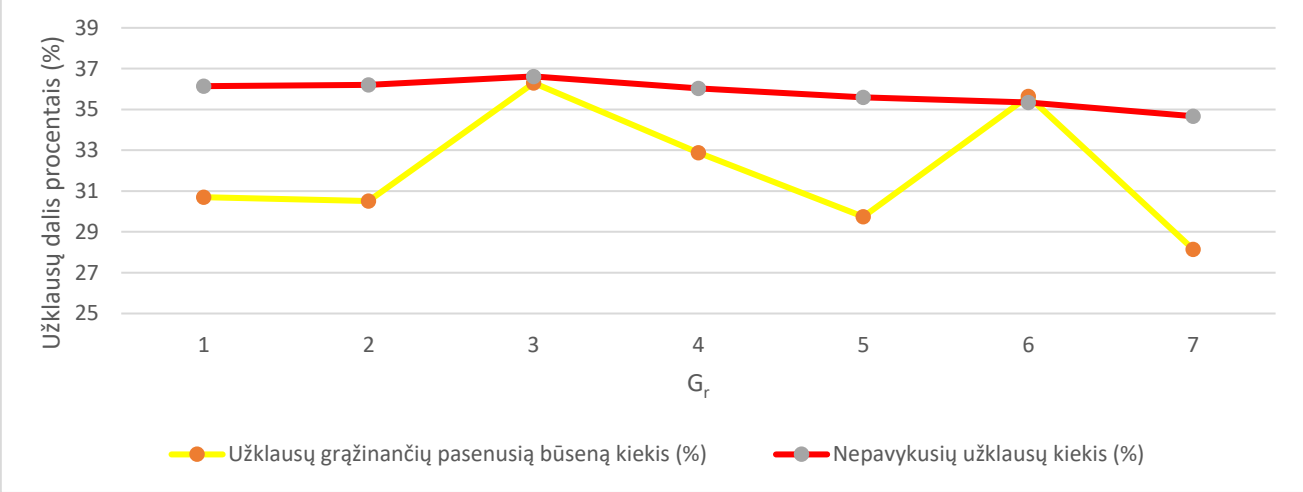
skaičių. Užklausų, kurios grąžina pasenusią būseną kiekis nuolat auga, kartu su didėjančiu G_r. Taip atsitinka, nes didėjant dublikatų skaičiui, didėja ir ryšių skaičius, kuriais būsena yra bendrinama. Tikimybė, kad nors vienas iš dublikatų einamuoju laiko momentu bus atskirtas nuo likusiųjų serviso komponentų laikui R_{dpa} yra skaičiuojama taip: $P = R_{dp} \times G_r$.

9.2.4. Būsenos bendrinimo tipo įtakos užklausų trukmei bei vientisumui tyrimas

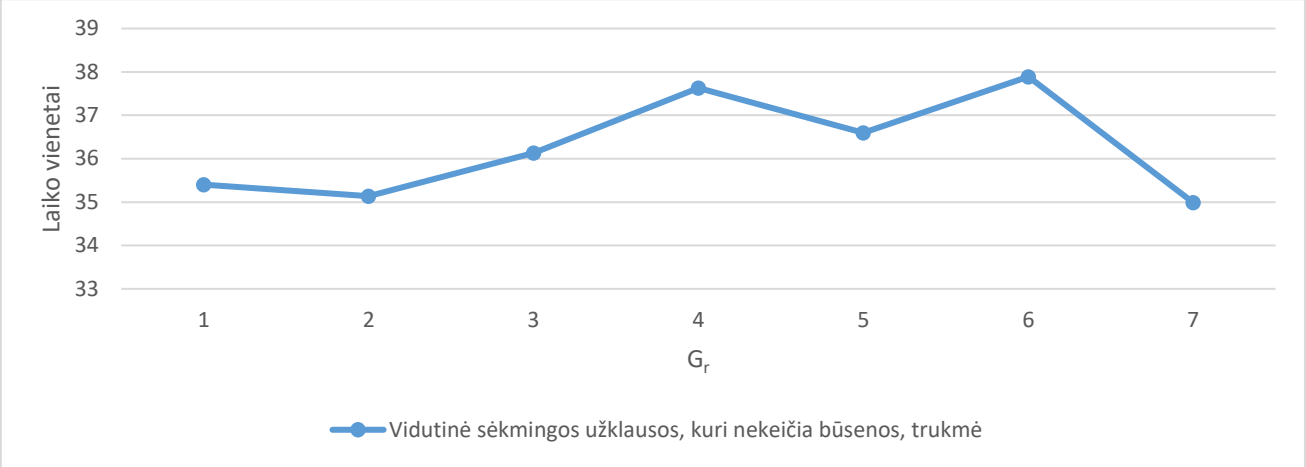
Šiame skyriuje bus tiriama serviso dublikatų skaičiaus įtaka vidutinei užklausų trukmei bei vidutiniam procentiniam kiekiui užklausų, kurios grąžina pasenusią būseną, kuomet parametras G_b = „multi-šeimininkas“. Kiti parametrai:

- Naudotojo, kurio užklausos nekeičia būsenos parametrai $V_{umax} = 7$ bei $V_{umin} = 5$;
- Pavyzdinio ryšio tarp komponentų parametrai $R_{dp} = 0{,}01$; $R_{dpa} = 1$.

Gauti rezultatai:



18 pav.: Užklausų grąžinančių pasenusią būseną kiekio bei nepavykusių užklausų kiekio priklausomybė nuo dublikatų kiekio G_r.



20 pav.: Sėkmingos užklausos, kuri nekeičia būsenos, trukmės priklausomybė nuo dublikatų kiekio G_r.

Nei apie vieną iš šių grafikų nebūtų galima pasakyti, jog reikšmės tendencingai mažėja ar didėja, kuomet G_r parametras didėja. Pokyčiai pastebimi, tačiau gana nežymūs (19 pav. pateikiamuose grafikuose minimumo ir maksimumo reikšmių skirtumai ne didesni nei 5 proc., o 20 pav. grafiko minimumo ir maksimumo reikšmės skiriasi iki 3 laiko vnt.).

Šiuose rezultatuose galima įžvelgti keletą „multi-šeimininkas“ būsenos bendrinimo tipo savybių. Visų pirma, lyginant su „šeimininkas-tarnas“ tipu (bandymai atlikti su tokiais pat parametrais, todėl galima lyginti), pastebimas gana žymus užklausų grąžinančių pasenusią būseną kiekio padidėjimas, šiek tiek didesnis klaidų skaičius bei laiko vienetų, reikalingų įvykdyti užklausą, kuri nekeičia būsenos, trukmės išaugimas. Tačiau galima įžvelgti ir teigiamą savybę – elgsenos stabilumą nepriklausomai nuo G_r parametro. Tai reiškia, jog dublikatų skaičius gali smarkiai padidintas (su visomis iš to išplaukiančiomis teigiamomis savybėmis, tokiomis kaip atsparumas klaidoms ir pan.) išvengiant sumažėjusio vientisumo.

9.3. Tyrimų rezultatų bei išvadų apibendrinimas

Šių tyrimų rezultatas – bandymų metu gautos išvados apie architektūrinių sprendimų savybes esant tam tikroms, apibrėžtoms, sąlygoms. Atliekant bandymus ir analizuojant gautus rezultatus buvo įsitikinta, jog vertinant architektūrinius sprendimus siūlomu metodu, galima pastebėti savybes, kurias būtų sudėtinga nustatyti samprotaujant. Pvz. 9.2.2 skyriuje aprašyto bandymo metu, buvo nustatyta, jog nedidelis kiekis tinklo trūkių gali užtikrinti didesnį vientisumą, kuomet apkrova sistemai yra šiek tiek per didelė. Taip pat, galima tirti architektūrinių sprendimų ar stilių topologijų įtaką savybėms, kaip tai buvo atlikta 9.2.4 skyriuje, nuspėjant, kaip sistema elgsis kuomet pvz. bus didinamas

dublikatų skaičius: kada tai yra naudinga, o kada tai gali atnešti nepageidaujamų reiškinių (kurie buvo pastebėti 9.2.3 skyriuje).

Šiuose tyrimuose buvo vertinami SOA architektūriniu stiliumi paremti sprendimai vientisumo, klaidų kiekio bei užklausų trukmės, kurios nekeičia būsenos, atžvilgiu. Tai yra antrojoje PACELC teoremos dalyje minimos savybės, tarp kurių, kaip teorema teigia, reikia rinktis. Šią būtinybę rinktis iliustruoja bandymas atliktas 9.2.3 skyriuje. Tačiau realiose sistemose, veiksnių darančių įtaką yra labai daug, todėl tame pačiame 9.2.3 skyriuje galima pastebėti, jog teorema veikia tik tol, kol dublikatų skaičius nėra pernelyg didelis: kai apkrova komponentams (ypač „šeimininko“ komponentui) išauga dėl padidėjusio bendrinimo užklausų kiekio, tuomet ima mažėti vientisumas ir augti užklausų trukmė, kas prieštarautų teoremai. Tai taip pat parodo šiame darbe kurto metodo vertę: ja galima tyrinėti pvz. kaip teoremos veikia realiose sąlygose.

REZULTATAI BEI IŠVADOS

Rezultatai

Siekiant darbo tikslo, buvo gauti šie rezultatai:

- Pagal pasirinktus kriterijus metodui, kuris reikalingas darbo tikslo įgyvendinimui, buvo apžvelgtos ir įvertintos esamos metodikos bei metodai skirti programų sistemų savybių tyrinėjimui;
- Remiantis tais pačiais kriterijais įvardintais ankstesniame punkte, buvo sukurtas metodas, leidžiantis vertinti nefunkcines architektūrinių sprendimų bei stilių, kurie išskirsto komponentus tinkle, savybes;
- Sukurtas įrankis leidžiantis pasinaudoti metodu bei jį validuoti, metodas validuotas;
- Pateikti metodo panaudojimo pavyzdžiai, tirtos architektūrinių sprendimų savybės remiantis PACELC teoremoje apibrėžtais teiginiais.

Išvados

Sukūrus metodą bei įrankį leidžiantį jį taikyti, buvo atlikta validacija bei pateikti metodo panaudojimo pavyzdžiai, kurių metu išryškėjo metodo panaudojamumo galimybės. Pastebėta, jog metodą prasmingiausia naudoti siekiant analizuoti didesnės apimties, įvairialypius sprendimus, kurių veikimą sudėtinga nuspėti naudojantis stilių ir sprendimų savybių aprašymais bei samprotaujant. Šį teiginį įrodo 9.2 skyriuje pateikiamų bandymų rezultatai, kurių metu, tam tikrais atvejais, buvo sulaukta netikėtų reiškinių, kuriuos būtų labai sudėtinga prognozuoti. Taip pat, šis metodas gerai tinka vertinimui, kaip teorija veikia realiuose architektūriniuose sprendimuose. Tai buvo parodyta per PACELC teoremą: vertinta, ar teoremoje pateikiami teiginiai teisingi, kuomet architektūriniam sprendimui įtaką daro šalutiniai veiksniai, pvz. tokie kaip didėjanti sistemos komponentų apkrova, didinant dublikatų skaičių. Buvo pastebėta, kad esant šalutiniams veiksniams, teoremoje esantys teiginiai gali neatitikti tikrovės, kas nereiškia, jog teorema yra klaidinga, o tik tai, jog praktinėje sistemų inžinerijoje būtina įvertinti šalutinių veiksnių daromą įtaką, ką ir padeda atlikti šiame darbe pateikiamas metodas.

ŠALTINIAI

[Aba12] D. J. Abadi. Consistency Tradeoffs in Modern Distributed Database System Design. *Computer, Vol. 45, Issue 2, p. 37–42*, 2012. DOI: 10.1109/MC.2012.33.

[Amba] Scott W. Ambler. Agile Architecture: Strategies for Scaling Agile Development. http://www.agilemodeling.com/essays/agileArchitecture.htm, tikrinta 2018-04-14. 2001-2018.

[Ambb] Scott W. Ambler. Architecture Envisioning: An Agile Best Practice, http://agilemodeling.com/essays/initialArchitectureModeling.htm, tikrinta 2018-04-14. 2001-2018.

[BB98] P. Bengtsson, J. Bosch. Scenario-based Software Architecture Reengineering. *Fifth International Conference on Software Reuse*. IEEE, Victoria, Canada, 1998. DOI: 10.1109/ICSR.1998.685756.

[BCK03] L. Bass, P. C. Clements, R. Kazman. *Software Architecture in Practice (2nd Edition).* Addison-Wesley Professional, USA, 2003. ISBN: 0321154959.

[BF14] P. Bourque, R. E. Fairley. *Guide to the Software Engineering Body of Knowledge Version 3.0*. IEEE Computer Society, USA, Washington, 2014.

[BGH+02] K. S. Barber, T. Graser, J. Holt, G. Baker. Arcade: early dynamic property evaluation of requirements using partitioned software architecture models. *Requirements Eng.* Springer-Verlag, Berlin, Germany, 2003, p. 222–235. DOI: 10.1007/s00766-002-0159-4.

[Bre00] E. Brewer. Towards Robust Distributed Systems, *PODC Keynote*, http://pld.cs.luc.edu/courses/353/spr11/notes/brewer_keynote.pdf, tikrinta 2018-04-14. 2000.

[Bon02] E. Bonabeau. *Agent-based modeling: Methods and techniques for simulating human systems*. National Academy of Sciences, USA, Washington, 2002. DOI: 10.1073/pnas.082080899.

[Bur14] S. Burckhardt. *Principles of Eventual Consistency*. Microsoft Research, USA, Redmond, 2014. DOI: 10.1561/2500000011.

[CNY+00] L. Chung, B. A. Nixon, E. Yu, J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Springer US, USA, 2000. DOI: 10.1007/978-1-4615-5269-7.

[EFG+03] P. TH. Eugster, P. A. Felber, R. Guerraoui, Anne-Marie Kermarrec. The Many Faces of Publish/Subscribe. ACM Computing Surveys, USA, New York, 2003, p. 114–131. DOI: 10.1145/857076.857078.

[EMM07] G. Edwards, S. Malek, N. Medvidovic. Scenario-Driven Dynamic Analysis of Distributed Architectures. *FASE'07 Proceedings of the 10th international conference on Fundamental approaches to software engineering*. Portugal, Braga, 2007. DOI: 10.1007/978-3-540-71289-3_12.

[ERR11] N. Eftekhari, M. P. Rad, H. Alinejad Rokny. Evaluation and Classifying Software Architecture Styles Due to Quality Attributes. *Australian Journal of Basic and Applied Sciences,* 2011. ISSN: 1991-8178.

[Fie00] R. T. Fielding. Architectural styles and the design of network-based software architectures. Disertacija. University of California, Irvine, 2000. ISBN: 0-599-87118-0.

[GJN11] P. Gill, N. Jain, N. Nagappan. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. *SIGCOMM'11.* Toronto, Ontario, Canada, 2011. DOI: 10.1145/2043164.2018477.

[GL02] S. Gilbert, N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, Vol. 33, Issue 2, 2002, p. 51-59. DOI: 10.1145/564585.564601.

[GLM+08] A. Greenberg, P. Lahiri, D. A. Maltz, P. Patel, S. Sengupta. *Towards a Next Generation Data Center Architecture: Scalability and Commoditization*. Microsoft Research, Redmond, WA, USA, 2008.

[GS94] D. Garlan, M. Shaw. An Introduction to Software Architecture. *Advances in Software Engineering and Knowledge Engineering*. World Scientific Publishing Company, Singapore, 1994.

[Har11] N. B. Harrison. Improving Quality Attributes of Software Systems Through Software Architecture Patterns. Disertacija. University of Groningen, 2011. ISBN: 978-90-367-4893-3.

[HBR14] L. Happe, B. Buhnova, R. Reussner. Stateful component-based performance models. *Software and systems modeling*, Vol. 13, 2014, p. 1319–1343. DOI: 10.1007/s10270-013-0336-6.

[HR94] F. Hayes-Roth. *Architecture-based acquisition and development of software: Guidelines and recommendations from the ARPA domain-specific software architecture (DSSA) program*. Teknowledge Federal Systems, Palo Alto, CA, 1994.

[ICSA17] IEEE International Conference on Software Architecture. *Gothenburg, Sweden*, 2017.

[ISO11] International Organization for Standardization. ISO/IEC 25010:2011, 2011.

[ISR] Architecture-based Design. Institute for Software Research, University of California, Irvine. https://isr.uci.edu/architecture/archdesign.html, tikrinta 2018-04-15.

[Joh00] J. M. Johansson. On the impact of network latency on distributed systems design. *Information Technology and Management*, Vol. 1, 2000. DOI: 10.1023/A:1019121024410.

[KAB+96] R. Kazman, G. Abowd, L. Bass, P. Clements. Scenario-Based Analysis of Software Architecture. *IEEE Software*, Vol. 13, 1996. DOI: 10.1109/52.542294.

[KB12] F. Klügl, A. L. C. Bazzan. Agent-Based Modeling and Simulation, *Association for the Advancement of Artificial Intelligence*, Vol. 33, 2012. DOI: https://doi.org/10.1609/aimag.v33i3.2425.

[Kru99] P. Kruchten. *The Rational Unified Process: An Introduction.* Addison-Wesley Professional, USA, Boston, 1999.

[LRM+07] J. Lemmetti, M. Raatikainen, V. Myllärniemi, T. Männistö. Comparison of Software Architecture Design and Extreme Programming, *IFIP Central and East European Conference on Software Engineering Techniques*, 2007.

[Mar97] A. Maria. Introduction to Modeling and Simulation, *Proceedings of the 1997 Winter Simulation Conference.* Atlanta, USA, 1997.

[Mår06] F. Mårtensson. SOFTWARE ARCHITECTURE QUALITY EVALUATION APPROACHES IN AN INDUSTRIAL CONTEXT. Disertacija. Blekinge Institute of Technology, 2006.

[McC07] J. D. McCabe. *Network Analysis, Architecture, and Design, 3rd Edition*. Morgan Kaufmann Publishers, USA, Burlington, 2007.

[MHH+09] J.D. Meier, D. Hill, A. Homer, J. Taylor, P. Bansode, L. Wall, R. Boucher, Jr. A. Bogawat. *Microsoft Application Architecture Guide, 2nd Edition*. *Microsoft, USA, Redmond*, 2009.

[Mic06] B. M. Michelson. Event-Driven Architecture Overview. http://soa.omg.org/Uploaded%20Docs/EDA/bda2-2-06cc.pdf, tikrinta 2018-04-15. 2006.

[MN09] C. M. Macal, M. J. North. AGENT-BASED MODELING AND SIMULATION, *Proceedings of the 2009 Winter Simulation Conference.* IEEE, USA, New Jersey, 2009.

[Mik14] A. Mikoliūnas. Programų sistemų architektūrų tyrimas taikant agentais grįstą modeliavimą. Magistrinis darbas. Vilniaus Universitetas, 2014.

[NHK09] M. A. Niazi, A. Hussain, M. Kolberg. Verification & Validation of Agent Based Simulations using the VOMAS (Virtual Overlay Multi-agent System) Approach, *Proceedings of the Second Multi-Agent Logics, Languages, and Organisations Federated Workshops.* Italy, 2009.

[Ode02] J. Odell. Object and Agents Compared. *JOURNAL OF OBJECT TECHNOLOGY, ETH Vol. 1, No. 1,* 2002.

[Roy70] Winston W. Royce. MANAGING THE DEVELOPMENT OF LARGE SOFTWARE SYSTEMS, *Proceedings IEEE WESTCON*, 1970, p. 1-9.

[SCD+03] M. L. Scott, D. Chen, S. Dwarkadas, C. Tang. Distributed Shared State (position paper), *Proceedings of the The Ninth IEEE Workshop on Future Trends of Distributed Computing Systems.* http://www.cs.rochester.edu/~scott/papers/2003_FTDCS_IW.pdf, tikrinta 2018-04-15. 2003.

[Sch11] G. Schmutz. Event-Driven SOA: Events Meet Services, *Oracle Technology Network*, http://www.oracle.com/technetwork/articles/soa/schmutz-soa-eda-405955.html, tikrinta 2018-04-15. 2011.

[SEI10] Software Engineering Institute. What is your definition of software architecture? *Carnegie Mellon University*. https://resources.sei.cmu.edu/asset_files/FactSheet/2010_010_001_513810.pdf, tikrinta 2018-04-15, 2010.

[Sho16] J. Shore. Synchronous vs. asynchronous communication: The differences. http://searchmicroservices.techtarget.com/tip/Synchronous-vs-asynchronous-communication-The-differences, tikrinta 2018-04-15. 2016.

[Sil16] A. Siliūnas. Programų sistemų, kuriose yra komponentų su būsenomis, architektūrų tyrimas taikant agentais grįstą modeliavimą. Magistrinis darbas. Vilniaus Universitetas, 2016.

[SOA17] VERSIONONE. 11th annual STATE OF AGILE REPORT. VERSIONONE.COM, 2017.

[Toxi18] Toxiproxy. A TCP proxy to simulate network and system conditions for chaos and resiliency testing, https://github.com/Shopify/toxiproxy, tikrinta 2018-04-15. 2018.

[VDR00] M. E. R. Vieira, M. S. Dias, D. J. Richardson. Analyzing Software Architectures with Argus-I. *Proceeding ICSE '00 Proceedings of the 22nd international conference on Software engineering*. Ireland, 2000, p. 758-761.

[VDT+07] T. Verdickt, B. Dhoedt, F. D. Turck, P. Demeester. Hybrid Performance Modeling Approach for Network Intensive Distributed Software. *Proceedings of the 6th international workshop on Software and performance*. Argentina, Buenos Aires, 2007.

[Woo13] M. Woodside. Tutorial Introduction to Layered Modeling of Software Performance, *Carleton University*, http://www.sce.carleton.ca/rads/lqns/lqn-documentation/tutorialh.pdf, tikrinta 2018-04-15. 2013.

[WY12] Q. Wang, Z. Yang. A method of selecting appropriate software architecture styles: Quality Attributes and Analytic Hierarchy Process. Bakalauro darbas. University of Gothenburg, Chalmers University of Technology, 2012.

[XKM05] X. Xiang, R. Kennedy, G. Madey. Verification and Validation of Agent-based Scientific Simulation Models. *Proceedings of the 2005 Agent-Directed Simulation Symposium*. USA, San Diego, 2005.