KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PROGRAMŲ INŽINERIJOS KATEDRA

Konstantin Kolganov

Paskirstytų sistemų testavimo metodikos kūrimas ir tyrimas

Magistro darbas

Darbo vadovas
prof. E. Bareiša

Kaunas, 2010

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PROGRAMŲ INŽINERIJOS KATEDRA

Konstantin Kolganov

Paskirstytų sistemų testavimo metodikos kūrimas ir tyrimas

Magistro darbas

Recenzentas	Darbo vadovas
dr. A. Lenkevičius	prof. E. Bareiša
2010-05-	2010-05-

Atliko
IFM-4/2 gr. stud.
Konstantin Kolganov
2010-05-22

Kaunas, 2010

Santrauka

Automatinės įeigos kontrolės sistemos vis labiau ir labiau integruojasi į kiekvieną technologijos ir gyvenimo sritį. Jos be papildomo žmogaus įsikišimo gali kontroliuoti žmonių patekimą į patalpas ar pastatus. Jau gana seniai naudojamos sistemos, kuriose reikia per skaitytuvą perbraukti ID kortelę ar įvesti tam tikrą numerį (PIN – *Personal Identification Number* – kodą), kuris atrakina spyną. Biometrinės sistemos, veikiančios žmogaus fiziologijos pagrindu, atsirado palyginti nesenai, kai technologijų tobulėjimo lygis pasiekė pakankamą, kad būtų imanoma greitai nuskaityti ir apdoroti žmogaus biometrinius duomenis, tokius kaip, pvz., piršto atspaudus, akies rainelę, balso, veidą ar jų kombinaciją. Biometrinės atpažinimo sistemos yra patikimesnės nei, pvz., kodinės sistemos, nes sumažėja žmogiškojo faktoriaus problema. Pvz., ID kortelę galima pamesti, kodą gali sužinoti pašaliniai asmenys, žmonės kodus kartais linkę užsirašinėti, todėl taip informacija tampa dar labiau prieinama asmenims, kurie jos neturėtų žinoti. Biometrinių sistemų atveju prisiminti ar užsirašyti nieko nereikia. Tiesiog pakanka padėti pirštą ant skaitytuvo, pasakyti kokią nors frazę į mikrofoną, ar atsistoti prieš kameros objektyvą, o sistema atlieka visą likusį darbą.

Šiais technologijų laikais, kai informacinių sistemų plėtojimosi sparta senokai viršijo techninės įrangos galimybių plėtojimo spartą, kai informacinės technologijos plinta į dar prieš kelis metus buvusius, atrodo, visai nesuderinamais su IT įrenginius (intelektualūs šaldytuvai, mikrobangų krosnelės ir pan.), o pasaulyje nesustabdomai didėja informacinis poreikis, kūrėjai vis dažniau taikosi į paskirstytos architektūros sistemas. Tačiau paskirstytos sistemos nėra tokia jau ir naujovė: pirmosios paskirstytos sistemos atsirado kartu su kompiuterių tinklais, ir iki šiol neaišku, ar tinklai atsirado dėl poreikio paskirstytoms sistemoms, ar paskirstytos sistemos atsirado dėl kompiuterinių tinklų suteikiamų privalumų. Viena aišku: šios dvi technologijos nuo to laiko tapo neatsiejamos viena nuo kitos.

Šiame darbe nagrinėjamos paskirstytos sistemos ir jų testavimo ypatumai, taikomos testavimo metodikos ir kuom paskirstytų sistemų testavimas skiriasi nuo nepaskirstytų (*lokalių*) sistemų testavimo siekiant parinkti tinkamiausią metodiką ar jų kombinaciją. Tyrimas atliekamas pritaikius ir išanalizavus skirtingas testavimo metodikas paskirstytos sistemos *Veidą Atpažįstanti Įeigos Kontrolės Sistema* pagrindu. Ši sistema remiasi žmogaus biometriniais duomenimis, konkrečiau veido atvaizdu, kad nustatyti, ar vartotojas gali patekti į sistemos saugomas patalpas.

Abstract

Establishment and research of testing methodology for distributed systems

Automated access control systems are becoming increasingly integrated within any aspect of technology, be that IT or regular business appliances. Such systems can automatically control access to offices, premises or whole buildings. Systems, which require either PIN (*Personal Identification Number*), ID card or like to be used to authenticate one and allow access to protected area. Biometric systems, which are based on unique aspects of human physiology, came into wider usage only recently, when technological advances made such technology viable for wide application. Currently most biometric systems use fingerprints, iris, voice, face or a combination of them. Biometric access control systems are more reliable, because one cannot accidentally *forget* one's eye or finger at home, biometric information cannot be stolen, lost, is hard to counterfeit and it takes human factor out of the equation. Using biometrics is as simple as putting one's finger or eye to the scanner, saying phrase or just getting photographed by a camera, the systems handles everything else automatically.

Nowadays, when IT systems development speed has gone way ahead of hardware capabilities, when IT is rapidly expanding into devices until recently thought to be incompatible with such technology (refrigerators, air conditioners, microwave ovens, etc.) and the world's need for information is still accelerating, system developers even more frequently look up to distributed systems. However, distributed systems are not new: they have been around since invention of computer networks and there's still not clear, which idea influenced the other's emergence. Nonetheless, since then those two technologies have become inseparable.

This work deals with distributed systems and their testing peculiarities, applicable testing methodologies and differences between distributed and local systems' testing in pursuance of defining the most suitable method or combination of methods. Research is done by analyzing and applying different testing methodologies in testing of distributed system, known as *Face Recognizing Access Control System*. This particular system applies biometric data, particularly user's face image, to verify ones permission to enter protected area.

Turinys

1. Įvadas 7

2. Paskirstytų sistemų projektavimo ir testavimo metodikų analizė 8

2.1. Paskirstytų sistemų projektavimo metodų analizė 8

2.1.1. Kodėl paskirstytos sistemos? 8

2.1.2. Kas gi yra paskirstyta sistema? 10

2.1.3. Paskirstytos ar lygiagrečios sistemos? 12

2.1.4. Paskirstytų sistemų kūrimo problemos 14

2.1.5. Paskirstytų sistemų kūrimo metodikos 18

2.1.6. Paskirstytų sistemų kūrimo analizės apibendrinimas 19

2.2. Paskirstytų sistemų testavimo metodų analizė 21

2.2.1. Skaičiuojamieji modeliai (*computational models*) 21

2.2.2. Sinchronizacija: metodikos ir testavimas 24

2.2.3. Kūrimas testuojamumui (*design for testability*) 26

2.2.4. Pagrindinės testavimo metodologijos 28

3. Veidą atpažįstančios įeigos kontrolės sistemos analizė 31

3.1. Panašios sistemos 31

3.1.1. IITS FRS ACCESS 31

3.1.2. „NeoFace Control" 32

3.1.3. FxGuard Pro 34

3.2. Panašių sistemų palyginimas 35

4. Veidą atpažįstančios įeigos kontrolės sistemos projektavimo ypatumai 37

4.1. Architektūros tikslai ir apribojimai 37

4.2. Sistemos panaudojimo atvejų vaizdas 39

4.2.1. Standartiniai sistemos panaudojimo scenarijai 39

4.3. Didžiausi sistemos architektūros ypatumai 40

4.4. Komponentų detalizavimas 42

4.4.1. Kameros modulis 42
4.4.2. Spynos modulis 43
4.4.3. Administratoriaus valdymo skydelis 43
4.4.4. Pagrindinis valdymo modulis 44
4.4.5. Duomenų bazės modulis 45
4.5. Automatinis kodo generavimas 45
4.6. Sistemos dinaminis vaizdas 46
4.6.1. Bendradarbiavimo diagrama 46
4.6.2. Vartotojo atpažinimo ir įleidimo į patalpas sekų diagrama 47
5. Paskirstytos sistemos VAĮKS testavimo metodikų taikymas ir tyrimas 48
5.1. Testavimo metodikų aprašymas ir taikymas 48
5.1.1. Bendra testavimo metodika 48
5.1.2. Kameros modulio testavimo metodika 49
5.1.3. Spynos modulio testavimas 50
5.1.4. Administratoriaus skydelio testavimas 51
5.1.5. Pagrindinio valdymo modulio testavimas 51
5.1.6. Duomenų bazės modulio testavimas 52
5.1.7. Integracinis testavimas 53
5.1.8. Sistemos testavimas 54
5.2. Testavimo metodikų taikymo paskirstytų sistemų testavime tyrimas 55
5.2.1. Testavimo metodų vertinimo kriterijai (metrikos) 55
5.2.2. Testavimo metodikų pritaikomumas ir įvertinimas 56
5.2.3. Testavimo metodikų palyginimas 59
6. Išvados 61
7. Literatūros sąrašas 62

1. Įvadas

Automatinės įeigos kontrolės sistemos tampa neatsiejama technologijos ir gyvenimo dalimi. Jų pagalba be papildomo žmogaus įsikišimo galima kontroliuoti žmonių patekimą į patalpas ar pastatus. Jau gana seniai naudojamos sistemos, kuriose reikia per skaitytuvą perbraukti ID kortelę ar įvesti tam tikrą numerį (PIN – *Personal Identification Number* – kodą), kuris atrakina spyną. Kiek vėliau atsirado biometrinės sistemos, veikiančios žmogaus fiziologijos pagrindu. Pvz., piršto atspaudo skaitytuvai, akies rainelės skaitytuvai, balso atpažinimo bei veido atpažinimo sistemos. Biometrinės sistemos yra patikimesnės nei, pvz., kodinės sistemos, nes sumažėja žmogiškojo faktoriaus problema. Pvz., ID kortelę galima pamesti, kodą gali sužinoti pašaliniai asmenys, žmonės kodus kartais linkę užsirašinėti, todėl taip informacija tampa dar labiau prieinama asmenims, kurie jos neturėtų žinoti. Biometrinių sistemų atveju prisiminti ar užsirašyti nieko nereikia. Tiesiog pakanka padėti pirštą ant skaitytuvo, pasakyti kokią nors frazę į mikrofoną, ar atsistoti prieš kameros objektyvą, o sistema atlieka visą likusį darbą.

Šiame darbe nagrinėjama biometrinių įeigos kontrolės sistemų dalis, pagrįsta asmens veido atpažinimu. Per paskutinį dešimtmetį veido atpažinimo technologija tapo viena iš svarbiausių kompiuteriu atliekamų vaizdo apdorojimo sričių. Pagrindinis šios srities tyrimų tikslas yra išanalizuotų nejudantį vaizdą (nuotrauką) arba filmuojamą medžiagą ir lyginant ją su duomenų bazėje esančiais įrašais, veidą identifikuoti arba verifikuoti. Veido identifikavimu vadinamas procesas, kai iš duomenų bazės išrenkama galimų atitikmenų seka, o verifikavimo metu iš atrinktos aibės išrenkamas vienintelis tinkamas įrašas – asmens atitikmuo. Atliekant tyrimus yra koncentruojamasi ties tokiais subjektais, kaip veidų atpažinimas žmonėms esant lauke (t.y., kai veidai yra gana toli nuo kameros), keleto veidų, esančių viename kadre užfiksavimas ir atpažinimas, veido atpažinimus jam nesant pilnai pasuktam į objektyvą, supratimas, kodėl vyrus lengviau identifikuoti nei moteris, nagrinėjimas, kaip demografinės savybės įtakoja veido atpažinimą ir pan.

Automatinė įeigos kontrolės sistema, veikianti veido atpažinimo pagrindu, yra labai patogi naudoti, nes žmogui tiesiog reikia prieiti prie kameros ir palaukti keletą sekundžių, kol kamera jį nufotografuoja, ir sistema atlieka identifikavimo procesą.

Veido atpažinimo sistemų trūkumas yra tas, kad jos dar nėra 100% patikimos. Tai įtakoja įvairūs veiksniai: apšvietimas, veido pasukimas, lytis (moteris sunkiau identifikuoti nei vyrus), demografinė kilmė ir kt. Dėl to veido atpažinimu pagrįstose įeigos kontrolės sistemos naudojamos dubliuojančios sistemos, pvz., akies rainelės nuskaitymas, PIN kodo įvedimas ir pan. Žinoma, taikant dubliuojančią ne biometrinę sistemą, išlieka anksčiau minėta žmogiškojo faktoriaus

problema, tačiau dauguma identifikavimo atvejų yra sėkmingi, todėl žmogiškasis faktorius stipriai sumažėja, lyginant su atveju, jei būtų taikoma vien tik kodinė arba ID kortelėmis pagrįsta įeigos kontrolės sistema.

Šiame darbe nagrinėjama sistema yra skirta žmonių įleidimui į tam tikras patalpas ar pastatus, pvz., darbuotojų patekimo į įmonės patalpas kontroliavimui. Sistema susideda iš vaizdo kamerų, serverio (-ių) ir administravimo kompiuterio (-ių). Serveryje veikia nuotraukų apdorojimo algoritmas, saugoma DB. Per administravimo kompiuterį prieinamas serverio valdymas. Taip pat per jį į administratorius į DB gali įtraukti naujus įrašus su darbuotojų informacija, prieiti prie žurnalizavimo informacijos apie darbuotojų apsilankymo vietas ir laiką, kada darbuotojas ten buvo.

Šia tema informacijos daugiausia ieškota internete. Yra keletas specialiai vaizdo apdorojimo tyrimams sukurtų tinklapių. Bene išsamiausias yra „*Face Recognition Homepage*“, prieinamas adresu http://www.face-rec.org/. Tinklapyje pateikiama naujienų apie pasiekimus veido atpažinimo tyrimuose, yra gausus rinkinys algoritmų, kurie panaudojami analizuojant asmens veidą, yra sąrašas gamintojų, kurie leidžia į rinką įrenginius veidams fotografuoti ar filmuoti ir juos iš gautos medžiagos išskirti. Taip pat prie kiekvienos temos pateikiama nuorodų į kitus šaltinius internete šia tema. Yra skiltis su nuorodomis į duomenų bazes, saugančias veidų atvaizdus. Šios DB naudojamos testavimo tikslais.

2. Paskirstytų sistemų projektavimo ir testavimo metodikų analizė

2.1. Paskirstytų sistemų projektavimo metodų analizė

2.1.1. Kodėl paskirstytos sistemos?

Šiais laikais, kai informacinių sistemų plėtojimosi sparta senokai viršijo techninės įrangos galimybių plėtojimo spartą, o pasaulyje nesustabdomai didėja informacinis poreikis, kūrėjai vis dažniau taikosi į paskirstytos architektūros sistemas. Tačiau paskirstytos sistemos nėra naujovė. Pirmosios paskirstytos sistemos atsirado kartu su kompiuterių tinklais, ir iki šiol neaišku, ar tinklai atsirado dėl poreikio paskirstytoms sistemoms, ar paskirstytos sistemos atsirado dėl kompiuterinių tinklų suteikiamų privalumų. Tačiau iki paskirstytų sistemų atsiradimo buvo lygiagrečios sistemos, kai viename kompiuteryje keli procesai bendrauja pasitelkdami pranešimais, tačiau dalinasi bendra atmintimi. Tokio tipo architektūros pirma atsirado XX a. 7-ame dešimtmetyje. Pirmosios kompiuteriniais tinklais paremtos sistemos buvo sukurtos XX a. 8-ame dešimtmetyje ir nuo to laiko įsitvirtino srityse, kur reikalingos didelės informacijos apdorojimo apimtys (matematiniai

skaičiavimai, aplinkos ir biologinis modeliavimas, ekonominis ir finansinis modeliavimas, duomenų surinkimas ir apdorojimas, atvaizdų apdorojimas, ir daug kitų). Tačiau didėjant visuomenės poreikiui ir kitos sritys pradėjo taikyti paskirstytas sistemas.

Paskirstytos sistemos turi šiuos pranašumus prieš lokalias:

- Daugiau pajėgumo už atitinkamą kainą: dažniausiai kylant procesorių galingumui kaina didėja neproporcionaliai, paskirstytose sistemose du procesoriai už dviejų procesorių kainą gali atlikti dvigubai daugiau darbo, kai dvigubai galingesnis procesorius gali kainuoti nuo kelių iki keliolikos kartų brangiau.
- Paskirstant sistemos komponentes geografiškai yra naudinga: suteikiama galimybė naudotis sistemos paslaugomis nesant fiziškai šalia sistemos (bankomatai, nuotolinė kontrolė).
- Sąveikaujanti komunikacija ir pramoga: žaidimai ir bendravimas nuotoliniu būdu, elektroninis paštas, ip-telefonija, greiti pranešimai ir t.t.
- Nutolęs turinys: tinklalapiai, muzikos ir video parsisiuntimas, ip-televizija, interaktyvi televizija, failų saugyklos, duomenų bazės ir t.t.
- Mobilumas: bevielės komunikacijos, mobilieji telefonai, nešiojami kompiuteriai ir pan.
- Padidintas patikimumas: servisų dubliavimas ir pan.
- Palaipsninis didėjimas.[8]

Šiuo metu didžioji dauguma sistemų, atliekančių dideles skaičiavimų apimtis, yra paremtos paskirstytų sistemų architektūra. Geriausi paskirstytų sistemų pavyzdžiai: telekomunikacijos tinklai, kompiuteriniai tinklai (*www, p2p, internet, ethernet*), paskirstytos duomenų bazės, tinklų failų sistemos (*network file systems*), paskirstytos duomenų apdorojimo sistemos, tokios kaip bankinės ar oro linijų rezervavimo sistemos, netgi realaus laiko sistemos, kaip lėktuvo valdymo sistemos, gali būti paskirstytos. Pagrinde paskirstytos sistemos taikomos tuomet, kai reikia atlikti didesnį skaičiavimų kiekį bet nenorima investuoti į galingesnę kompiuterinę sistemą (labai brangi arba net gali neegzistuoti, pasiekiama apjungiant kelis pigesnes bet silpnesnes sistemas į vieną klasterį), kai reikia patikimesnės sistemos (pasiekiama dubliuojant kritines sistemos dalis paskirstytoje architektūroje, taip išvengiant vieno trikčio taško, angl. *single point of failure*, problemos) arba tiesiog norima išlaikyti galimybę vėliau išplėsti sistemą pagal poreikius.

Nepaisant tokio paskirstytų sistemų paplitimo jų kūrimas ir testavimas vis dar kelia daug rūpesčių kūrėjams. Paskirstytų sistemų tema yra rengiami du kasmetiniai simpoziumai, *Symposium on Principles of Distributed Computing (PODC)*, rengiamas Amerikoje nuo 1982 m., ir jo

europietiškas analogas *International Symposium on Distributed Computing (DISC),* rengiamas nuo 1985 m. [1][4][5]

Pagrindinės paskirstytų sistemų problemas (jos bus aptariamos vėlesniuose skyriuose):

- Paskirstytų sistemų kūrimas yra sudėtingas: paskirstyta architektūra reikalauja itin atidaus kūrėjų darbo;
- Operacinių sistemų paskirstymas: šiuo metu pasaulyje egzistuoja begalės operacinių sistemų skirtų tiek namų kompiuteriams, tiek profesionalioms darbo stotims, serveriams, dedikuotoms paskirstytoms sistemoms ir t.t.;
- Programavimo kalbų pasirinkimas: programavimo kalbų pasirinkimas tik didėja, pradedant nuo senų, bet dar naudojamų ir plačiai paplitusių C/C++, Basic, Pascal/Delphi, naujų ir vis dar plėtojamų Java, C#, J#, F#, Visual Basic .NET bei siauriau specializuotų php, asp, ruby, python ir kitų;
- Sistemos efektyvumas – paskirstyta sistema ne visada gali būti efektyvesnė už analogišką lokalią sistemą, ypač jei paisoma paskirstytos sistemos kūrimo sunkumų;
- Patikimumas – sukurti patikimą paskirstytą sistemą nepalyginamai sunkiau nei analogišką lokalią, kurioje mažiau taškų, kur gali atsirast problemų ir mažesnė priklausomybė nuo nepatikimų komponentų (tokiu kaip tinklas);
- Administravimo problemos – paskirstytą sistemą yra sunkiau prižiūrėti, nes ji ne tik programiškai yra paskirstyta, bet tokios sistemos neretai gali būti paskirstytos ir geografiškai – pavyzdžiui, duomenų bazės serveris įmonės duomenų centre, terminalai išdėstyti po visas įmonės patalpas (o jei įmonė didelė, turi filialu?) ir panašios problemos;
- Tinklo problemos – ryšio trikčiai, duomenų praradimas, vėlinimas;
- Saugumas – paskirstytos sistemos saugumu tenka daugiau rūpintis, nes ji turi daugiau pažeidžiamų vietų dėl savo specifikos.[8]

2.1.2. Kas gi yra paskirstyta sistema?

Paskirstytą sistemą sunku nusakyti prieš tai nepaminėjus kitų susijusių dalykų, todėl pateiksiu kaskadinį paskirstytos sistemos aprašymą:

Programa (*program*) – tai mašininis kodas, kurį rašo programuotojas ir vėliau sukompiliuoja speciali programinė įranga, vadinama kompiliatoriumi;

Procesas (*process*) – tai, kas gaunama paleidus programą;

Pranešimas (*message*) – naudojamas bendravimui (komunikacijai) tarp procesų;

Paketas (*packet*) – pranešimo fragmentas, kuris gali būti siunčiamas laidais ar kitu būdu keliauti tarp kompiuterių;

Protokolas (*protocol*) – specializuotas pranešimų formato aprašas ir taisyklės, kurių procesai privalo laikytis, kad sėkmingai keistis pranešimais;

Tinklas (*network*) – tai speciali infrastruktūra, jungianti kompiuterius, darbo stotis (workstation), terminalus, serverius ir pan., jis susideda iš maršrutizatorių, sujungtų komunikacinėm jungtim (communication links);

Komponentas (*component*) – gali būti procesas arba techninės įrangos dalis, reikalinga procesui dirbti, palaikantis bendravimą tarp procesų, duomenų saugojimą ir pan.;

Paskirstyta sistema (*distributed system*) – programinė įranga, pasitelkianti įvairius protokolus kad koordinuoti daugelio procesų veiksmus tinkle tokiu būdu, kad visi komponentai dirbtu drauge kad atliktų vieną ar nedidelį rinkinį glaudžiai susijusių darbų (*tasks*).

> *You know you have a distributed system when the crash of a computer you've never heard of stops you from getting any work done.*
>
> *– Leslie Lamport*

Bendru atžvilgiu sistemas galima suskirstyti į keturias grupes pagal jų technines charakteristikas (instrukcijų srautus ir duomenų, kuriuos šios instrukcijos apdoroja):

- **SISD** (*single instruction-single data*) – tradicinė vienprocesorė sistema, kai viena instrukcija apdoroja vieną duomenų vienetą.
- **SIMD** (*single instruction-multiple data*), dar žinoma kaip vektorinis skaičiavimas, kai viena instrukcija naudojama atlikti veiksmai su tam tikrais duomenų rinkiniais (vektoriais). Dažniausiai naudojama duomenų srautų apdorojime. Pavyzdžiai būtų APU (*attached processor unit*) CELL procesoriuje, Intel SSE instrukcijų plėtinys, PowerPC AltiVec (*velocty engine*).
- **MISD** (*multiple instructions, single data*): dažniausiai nenaudojama ir neturinti prasmės sistema. Kartais panaudojama dubliuojančiųjų sistemų klasifikavimui.
- **MIMD** (*multiple insturctions, multiple data*): Vienu metu skirtingiems duomenų rinkiniams taikomos skirtingos instrukcijos – lygiagrečios sistemos (daugiaprocesorinės ar daugiakomponentės) ir paskirstytos sistemos.[8]

Kodėl yra kuriamos paskirstytos sistemos? Yra begalė tokios sistemos privalumų, tokių kaip galimybė prijungti nutolusius vartotojus atviru ir išplečiamu (*scalable*) būdu. Sakydamas *atviru* turiu mintyje, kad kiekvienas komponentas yra visada pasirengęs sąveikauti su kitais

komponentais. Sakydamas *išplečiamu*, turiu mintyje, kad sistema lengvai gali būti pakeista, kad prisitaikytų prie vartotojų, resursų ar skaičiavimo esybių (*computing entity*) pokyčių.

Taigi, paskirstyta sistema gali būti daug didesnė ir žymiai galingesnė dėl ją sudarančių komponentų galimybių nei atskirų (stand-alone) sistemų kombinacijos. Tačiau tai nelengva – tam, kad paskirstyta sistema būtų naudinga, ji turi būti patikima. Tai yra sudėtingas tikslas dėl bendravimo tarp vienu metu dirbančių komponentų painumo.

Tam, kad būtų patikima, paskirstyta sistema privalo turėti šias charakteristikas:

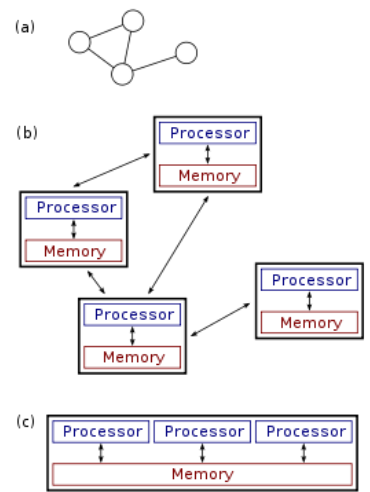
- Gedimų tolerancija (fault-tolerance): sistema gali atsigauti po komponentų trikčių neatlikdama netinkamų veiksmų.
- Pasiekiamumas (availability): sistema turi ir toliau teikti paslaugas nepaisant kai kurių komponentų gedimų.
- Atgautinumas (recoverability): sugedę komponentai turi sugebėti pasileisti iš naujo ir prisijungti prie sistemos po to, kai gedimo priežastis buvo pašalinta.
- Nuoseklumas (consistency): sistema turi gebėti koordinuoti daugelio komponentų veikimą nepaisant lygiagretumo (concurrency) ir gedimų. Tai pabrėžia sistemos galimybė veikti kaip nepaskirstytai sistemai.
- Išplečiamumas (scalability): Sistema turi veikti korektiškai net jei vienas jos aspektas yra išplėstas iki didesnės apimties. Pavyzdžiui, padidėjus tinklui, kuriame veikia sistema, gali padaugėti tinklo trikčių, kurie gali sutrikdyti neišplečiamos (non-scaleable) sistemos. Analogiškai, gali padidėti vartotojų ar serverių skaičiui, arba bendras sistemos apkrautumas. Išplečiamai sistemai tai neturėtų turėti reikšmingos įtakos.
- Nuspėjamas veikimas (predictable performance): galimybė pateikti norimus rezultatus laiku.
- Patikima (secure): sistema autentifikuoja prieigą prie paslaugų ir duomenų.[1]

2.1.3. Paskirstytos ar lygiagrečios sistemos?

Terminai „paskirstytos sistemos“ ir „lygiagrečios sistemos“ turi daug bendro ir nėra aiškių ribų tarp jų. Ta pati sistema gali būti charakterizuota ir kaip paskirstyta ir kaip lygiagreti: procesoriai tipiškoje paskirstytoje sistemoje veikia lygiagrečiai. Lygiagreti sistema gali būti traktuojama kaip labai glaudžiai susijusi paskirstyta sistema ir paskirstyta sistema gali būti traktuojama kaip laisvesnė lygiagrečios sistemos forma. Tačiau įmanoma klasifikuoti sistemas kaip lygiagrečias ar paskirstytas pagal šiuos kriterijus:

- Lygiagrečiose sistemose visi procesai turi priėjimą prie bendros (shared) atminties ir ši atmintis gali būti naudojama informacijos mainams tarp procesų.
- Paskirstytose sistemose kiekvienas procesas turi nuosavą atskirą atmintį ir koordinacija atliekama pranešimų mainais tarp procesų.

Paveiksliukas dešinėje iliustruoja skirtumus tarp paskirstytų ir lygiagrečių sistemų, Dalis (a) atvaizduoją tipinį paskirstytos sistemos atvaizdavimą grafu, kur kiekviena grafo viršūnė vaizduoją atskirą paskirstytos sistemos komponentą, o briauna – komunikacinę jungtį tarp komponentų. Dalis (b) atvaizduoja tą pačią sistemą detalesniu vaizdu: kiekvienas procesas turi nuosavą procesorių ir atmintį, o informacijos mainai tarp procesų vykdomi per komunikacines jungtis. Dalis (c) atvaizduoja lygiagrečią sistemą, kur kiekvienas procesas dalinasi atmintimi su kitais procesais. [2]

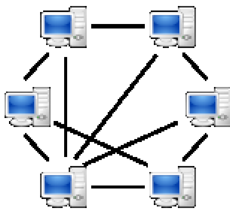


(a) Ir (b) – paskirstytos sistemos;
(c) – lygiagreti sistema

1 pav. ***Paskirstytos ir lygiagrečios sistemos.***

Galimi paskirstytų sistemų modeliai:

- **Centralizuotas modelis**: be tinklo, pasitelkianti tradicinę laiko dalijimosi (*time-sharing*) technologiją, vartotojų terminalai jungiami tiesiogiai prie sistemos. Sistemą sudaro vienas arba daugiau procesorių. Procesorių kiekis yra ribojantis faktorius, dėl kurio sistema yra sunkiai plečiama. Tokiu modeliu buvo pagrįstos senosios skaičiavimų sistemos.
- **Kliento-serverio modelis**: tokią sistemą sudaro serveriai ir klientai. Serveriai teikia klientams paslaugas (servisus). Aiški hierarchija tinkle, dažniausiai paplitusi architektūra internete.
- **Darbo vietos (*workstation*) modelis**: kliento-serverio architektūros atmaina, kai vienas klientas vienu metu naudojamas tik vieno vartotojo.
- **Lygiarangė (*peer-to-peer, p2p*) sistema:** sistemos architektūra, sudaryta iš dviejų arba daugiau procesų, kurie keičiasi duomenimis naudodami tą pačią arba to paties tipo programinę įrangą. Tokioje sistemoje resursų (atminties, tinklo, procesoriaus) naudojimas yra tolygiai paskirstytas tarp visų procesų, kurie gali naudoti ir vienas kito resursus. Tokioje sistemoje tinkle nėra paskirtų (*dedicated*) serverių ar klientų.



2 pav. ***Lygiarangės (peer-to-peer) sistemos pavyzdys.***

- **Procesoriaus fondo modelis (*processor pool model*):** šis sistemos modelis sprendžia problemą, ka daryti su komponentais, veikiančiais tuščia eiga (*idle*). Tokie resursai galėtų būti paskirstyti kitiems procesams, kurie reikalauja daugiau resursų, kad nebūtų veltui švaistomi sistemos resursai. Alternatyviai, tokia sistema gali sukurti procesorių/komponentų rinkinį, kuriam būtų galima paskirti darbus pagal reikiamybę.
- **Tinklelio (*grid*) modelis:** sistemos resursai paskirstomi taip, kad vartotojai turėtų vientisą (*seamless*) priėjimą prie resursų (skaiciavimo, duomenų saugyklos, tinklo pralaidumo). Tokia sistema susideda iš daugelio panašių komponentų, sujungtų tarpusavyje tokiu būdu, kad jie vienu metu dalintųsi tą pačią užduotį, t.y. vienas komponentas atlieka savą, maža dalį bendro darbo. Tokio tipo sistemos dažniausiai tinkamos daug skaičiavimų reikalaujantiems procesams vykdyti (tokie kaip vieni iš populiariausių *grid computing* pavyzdžių SETI@Home arba Folding@home projektai).

2.1.4. Paskirstytų sistemų kūrimo problemos

Pagrindinė paskirstytų sistemų kūrimo problema yra būtinybė atsižvelgti į klaidų kilimo galimybę. Trikčius pagrinde galima suskirstyti i dvi kategorijas: programiniai (*software*) ir techniniai (*hardware*). Techniniai trikčiai buvo dominuojanti problema iki XX a. 9-o dešimtmečio, tačiau nuo to laiko vidinės techninės įrangos kokybė ir patikimumas nepaprastai padidėjo. Sumažėjęs šilumos išskyrimas ir energijos suvartojimas dėl sumažėjusių grandinių, lustų jungčių ir laidų kiekio sumažėjimas, aukštos kokybės gamybos technikos – viskas sąlygojo teigiamą pokytį techninės įrangos patikimume. Šiandien problemos dažniausiai siejamos su jungtim ir mechaniniais įrenginiais, t.y. tinklo trikčiais ir diskų gedimais.

Programinės įrangos trikčiai yra lemianti problema paskirstytose sistemose. Net kruopščiai testuojant programinės klaidos sudaro esminę dalį sistemos prastovų priežasčių sąraše (apytiksliai 25-35%).[1]

Programinės klaidos (*bugs*) sistemose gali būti suskirstytos į kelias pagrindines kategorijas:

- ***Heisenbug***: Klaida, kuri paslaptingai išnyksta ar pakeičia savo charakteristikas kai ją bandoma stebėt ar ištirti. Dažniausiai pasitaikantis tokios klaidos pavyzdys yra klaida, kuri pasireiškia tik išleidimui skirtoje programos versijoje, bet ne tada kai yra tiriama derinimo režimu. Pats pavadinimas *heisenbug* yra kalambūras iš kvantinės fizikos termino „Heizenbergo neapibrėžtumo principas“. Šis terminas dažniausiai (tačiau nevisai teisingai) nusako stebėtojo efektą matavimams vien dėl to, kad jie stebi objektą (tai, tiesą pasakius, yra žinoma kaip *stebėtojo efektas* ir dažnai painiojamas su Heizenbergo neapibrėžtumo principu).

- ***Bohrbug*:** Programinės įrangos klaida (pavadinta pagal Boro atominį modelį), kuri, priešingai nei *heisenbug* neišnyksta ir nekeičia savo charakteristikų kai yra tiriama. Šio tipo klaidos dažniausiai patikimai pasireiškia esant tam tikroms, aiškiai apibrėžtoms, aplinkybėms.
- ***Mandelbug*:** Programinės įrangos klaida (pavadinta pagal fraktalų inovatorių Benoît Mandelbrot) yra klaida, kurios priežastys yra tiek sudėtingos, kad jos elgsena atrodo chaotiška. Dar šiuo terminu kartais vadinamos klaidos, kurių priežastys yra tiek sudėtingos, kad nėra praktinio šios klaidos pataisymo būdo. Geras šios klaidos pavyzdys yra klaida, kylanti iš fundamentalios sistemos architektūros klaidos.
- ***Schroedinbug*:** Klaida, kuri pasireiškia tik tada, kai kas nors išstudijuoja programos išeities kodą ar panaudoja programą neįprastu būdu. Nuo tada programa nustoja tvarkingai veikusi visiems kol klaida nesutvarkoma. Šis terminas pirma sykį pasirodė 2.9.9. *Jargon File* versijoje, išleistoje 1992 m. balandį. Pavadintas pagal *Shriodingerio katės* mąstymo eksperimentą. Gerai parašyta programa veikianti patikimoje sistemoje turėtų sekti determinizmo principais, ir todėl stebėjimo atvejis (programos sugadinimas jos teksto perskaitymu) negali turėti įtakos programos veikimui. Klaidingo kodo sutvarkymas dažniausiai yra svarbesnis nei supratimas, koks gi paslaptingas aplinkybių rinkinys priversdavo ši kodą veikti tvarkingai (arba bent jau sukeldavo jo teisingo veikimo įspūdį) ir kodėl jis staiga nustojo veikęs. Dėl šios priežasties šio tipo klaidos dažniausiai lieka nesuprastos. Jei šio tipo klaidos yra ištiriamos pakankamai detaliai jos gali būti perklasifikuojamos kaip *bohrbug, heisenbug* ar *mandelbug*.

Klaidos yra pagrindinis skirtumas tarp paskirstytų ir lokalių sistemų, todėl kuriant paskirstytas sistemas reikia atsižvelgti į klaidų neišvengiamumą. Įsivaizduokite, kad užduodate žmonėms klausimą „jei tikimybė, kad kažkas nutiks yra 1 iš 10^{13}*, kaip dažnai tai nutiks?“. Eilinis žmogus atsakytų „niekada.“. Tai yra be galo didelis skaičius žmogiškaja prasme, tačiau jei jūs užduotumėte tą patį klausimą fizikui, ji atsakytų „Visada. Kubiniame metre oro tokie dalykai nutinka pastoviai.“*

Kurdamas paskirstytas sistemas, privalai sakyti „klaidos vyksta pastoviai“. Tik tokiu būdu kurdamas paskirstytą sistemą tu kuri ją ateičiai. Tai svarbiausias rūpestis. Ką reiškia kūrimas ateičiai? Klasikinė problema yra dalinis triktis. Tarkim, aš siunčiu pranešima tau, ir besiunčiant nutinka ryšio triktis. Tokiu atveju yra du galimi variantai: pranešimas nuėjo iki tavęs, o tada ryšys sutriko, ir aš tiesiog negavau atsakymo; kitas variantas yra kai tinklas sutriko prieš pranešimui ateinant iki tavęs.

Taigi, jei aš taip ir negavau atsakymo, kaip man sužinoti, kuris iš dviejų variantų nutiko? Aš negaliu to nustatyti galų gale tavęs nesuradęs. Tinklas privalo būti suremontuotas ar tu turi ateiti, nes gal būt ne ryšys sutriko o tu mirei. Kaip tai keičia būdą, kaip aš kuriu dalykus? Viena aišku, tai prideda daugiklį sudėtingumo reikšmei, kadangi kuo daugiau veiksmų aš galiu atlikti su jumis, tuo apie daugiau būdų, kaip atsistatyti po trikčio, man tenka galvoti.

Tai nėra tas klausimas, kuris užduodamas kuriant lokalias sistemas. Jūs kviečiate objektą ir metodą. Jūs neklausiate „ar kvietimas atvyko?“. Toks klausimas neturi prasmės. Tačiau tai yra būtent tas klausimas paskirstytų sistemų kūrime. [6]

- ***Mėnulio fazės klaida:*** mėnulio fazė neretai yra pavadinama kaip kvaila priežastis, dėl kurios kyla klaida, dažniausiai kai kūrėjas yra susierzinęs dėl nuolatinio šios klaidos priežasties ieškojimo. *Jargon File* nurodo du retus atvejus, kai iš klaidos tikrųjų buvo sukeltos mėnulio fazės laiku. Tačiau bendru atveju programos, kurios turi nuo laiko priklausantį veikimą yra labiau pažeidžiami su laiku susijusioms problemoms. Jos gali kilti per tam tikrą laiko tarpą, kaip keliamieji metai arba kai procesas dirba dienos, mėnesio, metų ar tūkstantmečio (*pvz., 2000 metų klaida*) pasikeitimo momentu.
- ***Statistinė klaida:*** ši klaida dažniausiai pasireiškia tose kodo vietose, kurios turėtų generuoti atsitiktinę arba pseudo-atsitiktinę išvestį. Jas neįmanoma nustatyti patikimai atlikus kelis bandomuosius paleidimus, nes jos ir taip turėtų generuoti atsitiktinius rezultatus. Klaidingas veikimas pasireiškia tik atlikus daugelį paleidimų ir atlikus statistinę rezultatų analizę. Pavyzdžiui, kodas turėtų generuoti atsitiktinius taškus, tolygiai pasiskirsčiusius sferos paviršiuje, tačiau atlikus daug paleidimų gaunama, kad vienoje sferos pusėje yra žymiai daugiau taškų nei kitoje. [7]

Paskirstytose sistemose dažniausiai pasitaiko *heisienbug* ir *mandelbug* tipo klaidos. Taip yra dėl daugelio išorinių parametrų, sąlygojančių sistemos veikimą, bei sudėtingumą sistemos kūrėjui suvokti rišlų ir išsamų paskirstytos sistemos veikimo vaizdą.

Pasinagrinėkime galimų klaidų, kurios gali kilti paskirstytoje sistemoje, tipus:

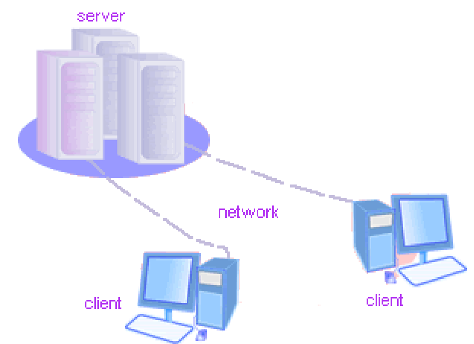
- Nustojimo veikti klaida (*halting failure*): komponentas tiesiog nustoja veikęs. Nėra jokio būdo nustatyti šį triktį apart neatsako laiko metodu, kai komponentas nustoja siusti „aš gyvas“ (*heartbeat*) pranešimus arba nereaguoja į užklausas. Kompiuterio „pakibimas“ yra viena iš šio tipo klaidų.
- *Fail-stop*: nustojimo veikti klaida, kai komponentas prieš sustodamas kokiu nors būdu praneša kitiems komponentams. Tinklinis failų serverio pranešimas klientams, kad jis tuoj nustos veikti yra tokio tipo klaida.
- Praleidimo trikčiai (*omission failures*): komponentas negali priimti/išsiųsti pranešimo dėl buferio talpos trūkumo, kas sukelia pranešimo atmetimą be kokio nors pranešimo siuntėjui ar gavėjui. Taip gali nutikti kai tinklo maršrutizatoriai perkrauti.
- Tinklo trikčiai: tinklo ryšio sutrikimai.
- Tinklo padalijimo trikčiai (*network partitioning failures*): kai tinklas skyla i dvi ar daugiau atskirų dalių, kurių viduje pranešimai gali būti siunčiami, bet jų neįmanoma išsiųsti už jų ribų. Taip dažnai nutinka dėl tinklo trikčių.

- Laiko sinchronizacijos klaida (*timing failure*): Sistemos laikinė savybė pažeidžiama. Taip dažnai nutinka jei skirtingose sistemose nesuderinti laikrodžiai, kai pranešimas užtrunka ilgiau nei paskirtas laiko tarpas ir pan.
- Įmantrūs trikčiai (*byzantine failures*): apjungia kelias trikčių ir klaidų rūšis, tokias kaip duomenų sugadinimas ar praradimas, trikčiai, sukelti piktų programų (*malicious software*) ir pan.[1]

Mūsų tikslas sukurti sistema, turinčia aukščiau išvardintas savybes (gedimų tolerancija, pasiekiamumas atgautinumas, nuoseklumas ir kt.), kas reiškia, kad mum reikia kurti atsižvelgiant į klaidas. Kuriant atsižvelgiant į klaidas būtina būti atsargiems ir nedaryti prielaidų apie sistemos komponentų patikimumą.

Kiekvienas, pirmą kartą kurdamas paskirstytą sistemą, padaro sekančias prielaidas. Šios prielaidos yra gerai žinomos paskirstytų sistemų srityje, kad jos dažniausiai vadinamos „8 paklydimai“ (*8 fallacies*).

1. Tinklas yra patikimas.
2. Gaištis (*latency*, t.y. laikas nuo užklausos duomenims išsiuntimo iki jų siuntimo pradžios) yra nulinė.
3. Tinklo pralaidumas (*bandwidth*) yra begalinis.
4. Tinklas yra saugus (*secure*).
5. Topologija (tinklas gali būti sukurtas remiantis skirtingomis topologijomis: žiedas, žvaigždė, magistralė, tinklinis ir kiti) nekinta.
6. Yra tik vienas administratorius.
7. Transportavimo kaštai yra nuliniai.
8. Tinklas yra vienalytis (*homogenous*, t.y. tinkle naudojamas tik vienas protokolas).[1]



3 pav. *Paprasčiausias paskirstytos sistemos modelis: serveris ir keli prie jo tinklu pasijungę klientai.*

*Kliento-serverio tipo sistemose **serveris** teikia tam tikras paslaugas, tokias kaip, pavyzdžiui, duomenų bazės ar biržos kursų informacija. **Klientas** jungiasi prie serverio ir naudojasi jo teikiamom paslaugomis (servisais) tam, kad pateikti vartotojui informaciją apie biržos kursus, tam tikras specifines rekomendacijas investicijom bei perduoti vartotojo komandas serveriui.*

*Yra begalės serverių tipų paskirstytose sistemose. Tai gali būti **failų serveris**, valdantis duomenų saugyklas, kuriose sukurtos failų kaupimo sistemos, **duomenų bazės** serveris saugo duomenų bazes ir teikia jų paslaugas kitiems tinklo kompiuteriams. **Tinklo vardų** serveris teikia simbolinių vardų vertimo į fizinius tinklo adresus paslaugas visiems tinklo kompiuteriams ir saugo informaciją apie tai, koks kompiuteris kokias paslaugas teikia.*[1]

2.1.5. Paskirstytų sistemų kūrimo metodikos

Sukurti patikimą sistemą, kuri veikia nepatikimame tinkle atrodo neįmanoma užduotis. Mes esame priversti tvarkytis su neapibrėžtumu. Kiekvienas procesas žino savo būseną, taipogi kiekvienas procesas žino kokioje būsenoje nesenai buvo kiti procesai. Tačiau procesas neturi galimybės sužinoti, kokioje konkrečiai būsenoje yra kiekvienas procesas duotu laiko momentu. Nes paskirstytose sistemose procesai neturi bendros atminties, be to jie neturi galimybės patikimų būdų atrasti klaidas, trikčius ar atskirti vietinį techninės ar programinės įrangos gedimą nuo ryšio trikčio.

Paskirstytos sistemos dizainas yra įdomi ir sunki užduotis. Kaip tai atlikti, jei mums negalima susidaryti prielaidų ir aplink yra tiek sunkumų? Pradėti reikia nuo akiračio apribojimo. Susitelksime ties viena specifine paskirstytų sistemų rūšimi, kuri naudoja kliento-serverio modelį ir taiko standartinius protokolus bendravimui. Pasirodo, standartiniai protokolai teikia pastebimą pagalbą patikimam žemo lygio bendravimui tinkle, kas pastebimai palengvina kūrimą.

Paskirstytose sistemose neretai pasitaiko keli vienodo tipo serveriai, pvz., keli failų serveriai. Terminas *paslauga* (*service*) naudojamas apibrėžti rinkinį tam tikro tipo serverių. Mes sakoma, kad įvyko *sietis* (*binding*), kai procesas, kuriam reikia prieigos prie tam tikros paslaugos susijungia su tam tikru serveriu, teikiančių tą paslaugą. Yra daug skirtingų *sieties taisyklių* (*binding policy*), nusakančių, kokia tvarka tam tikras serveris yra parenkamas. Pavyzdžiui, tvarka gali būti paremta lokališkumu (*locality*) (Unix NIS klientas visų pirmą ieško paslaugos toje pačioje mašinoje kaip ir jis pats), arba, ji gali būti paremta *apkrovos paskirstymu* (*load balancing*), kai pirmiausia jungiamasi prie mažiausiai apkrauto serverio ir stengiamasi išlaikyti vienodą serverių apkrovimą tinkle.

Paskirstyti servisai gali taikyti duomenų replikavimą (*data replication*), kai servisas palaiko vietines duomenų kopijas, taip leisdamas pasiekti tuos pačius duomenis lokaliai iš kelių skirtingų vietų ar tam, kad padidinti pasiekiamumą jei servisas nustojo veikti (*crashed*). *Kešavimas* yra viena iš susijusių koncepcijų, dažnai pasitaikančių paskirstytose sistemose. Sakoma, kad procesas *užsikeršavo* duomenis, jei jis palaiko lokalią šių duomenų kopiją tam atvejui, jei jų vėl prireiktų ir nereiktų kreiptis į serverį. *Kešo pataikymas* yra kai duomenų užklausa yra patenkinama iš lokalioje saugykloje turimų duomenų, be reikiamybės kreiptis į serverį. Pavyzdžiui, naršyklės taiko kešavimą, kad paspartinti prieigą prie dažniausiai naudojamų dokumentų.

Kešavimas yra panašus į duomenų replikavimą, tačiau išsaugoti lokalūs duomenys gali pasenti. Todėl reikia turėti mechanizmus, patikrinančius vietinių duomenų atitikimą serverių duomenims prieš juos panaudojant. Jei lokalūs duomenys yra aktyviai atnaujinami serverio, tokiu atveju keršavimas yra identiškas duomenų replikavimui. [8]

Kaip jau minėta anksčiau, komunikavimas tarp procesų privalo būti patikimas. Visi, tikriausiai, esate girdėję apie TCP/IP protokolą. *Interneto Protokolų* (*IP*) rinkinys (*suite*) yra rinkinys protokolų, įgalinančių komunikavimą internetu ir daugeliu kitų komercinių tinklų. *Transmission Control Protocol* (*TCP*) yra esminis rinkinio protokolas. Remiantis juom klientai ir serveriai gali kurti susijungimus tarpusavyje, per kuriuos jie gali keistis duomenimis paketais. Protokolas užtikrina patikimą pristatymą ir pristatymo eiliškumą iš siuntėjo gavėjui.[1]

Pagrindinės klaidų situacijos, į kurias tenka apdoroti:

- **Duomenų pakartotinas perdavimas dėl jų praradimo tinkle**: neretai sistema stengiasi pasiekti „daugiausiai vieną" duomenų persiuntimo bandymą. Blogiausiu atveju, nutikus pakartotino perdavimo situacijai, reikia pasistengti minimizuoti nuostolius dėl pakartotino duomenų perdavimo.
- **Serverio procesas „lūžta" užklausos metu**: Jei serverio procesas sustoja jam nespėjus užbaigti užduoties, sistema dažniausiai atsitaiso tvarkingai, nes klientas inicijuos pakartotiną užklausą serveriui atsistačius. Jei serveris nustoja veikti jau užbaigus vykdyti užduotį, bet anksčiau nei atsakymas yra perduodamas klientui, gali atsirasti pakartotinos užklausos dėl kliento pakartojimų.
- **Kliento procesas nustoja veikti nespėjęs gauti atsakymo**: Klientas paleidžiamas pakartotinai, o serveris atmeta užklausos rezultatų duomenis.

2.1.6. Paskirstytų sistemų kūrimo analizės apibendrinimas

Pasak Keno Arnoldo (*Ken Arnold*): „Paskirstytas sistemas būtina kurti tikintis jų veikimo trikčių". Reikia vengti daryti prielaidas, kad kitas komponentas sistemoje yra tam tikroje būsenoje. Klasikinis klaidos scenarijus yra kai procesas siunčia duomenis procesui, dirbančiam kitoje mašinoje (kompiuteryje). Procesas gauna duomenis, juos apdoroja ir nusiunčia duomenis atgal, darydamas prielaidą, kad kitas procesas yra pasiruošęs juos priimti. Tuo tarpu per tą laiką galėjo nutikti bet kas, ir siunčiantis procesas privalo būti pasiruošęs galimiems trikčiams.[6]

Reikia aiškiai apibrėžti galimus trikčių scenarijus ir nustatyti kokia yra kiekvieno jų atsitikimo tikimybė. Programinis kodas turi būti paruoštas susidoroti su labiausiai tikėtinomis situacijomis.

Tiek klientai, tiek serveriai privalo sugebėti susitvarkyti su neatsakančiais (*unresponsive*) siuntėjais/gavėjais.

Duomenų kiekis, numanomas siųsti tinklais, turi būti griežtai apgalvotas. Reikia stengtis kuo labiau sumažinti tinkle perduodamų duomenų kiekį.

Vėlinimas (*latency*) yra laiko tarpsnis tarp duomenų siuntimo užklausos iniciavimo ir duomenų siuntimo pradžios. Vėlinimo mažinimas neretai slypi klausime ar reikėtų daryti daug nedidelių duomenų užklausų ar vieną, bet didelį. Vienintelis patikimas būdas nustatyti atsakymą į šį klausimą yra eksperimentuoti ir rasti patikimiausią kompromisą.

Niekada negalime daryti prielaidos, kad duomenys, perduodami tinklu (ar netgi perduodami tarp diskų diskų masyve) yra tokie patys, kai jie atvyksta. Jei reikia nustatyti duomenų teisingumą, įterpkite duomenų validavimo procedūras sistemoje.

Kešavimas ir duomenų replikavimo strategijos yra metodikos, skirtos tvarkytis su komponentų būsenomis. Reikia stengtis mažinti komponentų, saugančių būsenas (*stateful*) paskirstytose sistemose, bet tai yra labai sudėtingas procesas. Būsenos informacija kartais yra saugoma kitoje vietoje, nei pats dirbantis procesas. Jei šie duomenys gali būti atstatyti kito komponento pagalba, tai – *kešas*. Kešai gerai tinkami mažinti riziką, susijusią su komponentų būsenų saugojimu, tačiau kešuoti duomenys gali tapti pasenę, todėl gali prireikti būdų, kad patikrinti, ar saugomi duomenys yra tinkami naudojimui, prieš juos panaudojant.

Jei procesas saugo informaciją, kurios neįmanoma rekonstruoti, tai gali iškilti problemų. Vienas galimų klausimų šiuo atveju yra „ar tu (komponentas) esi vienintelis trikties taškas?“ Man reikia su tavimi susisiekti dabar ir aš negaliu šiuo reikalu kalbėtis su kuo nors kitu. Taigi, kas nutiks, jei tau kas nutiks? Tam, kad susidoroti su šia problema tokie komponentai gali būti replikuojami. Replikavimo strategijos yra patikimas būdas mažinti procesų būsenos saugojimo riziką, tačiau iš čia kyla problemų: kas bus, jei aš pakeisiu vieno iš replikuotų komponentų duomenis ir tada bandysiu susisiekti su kitu? Ar spės mano pakeitimai pasiekti kitas komponento replikas laiku? Kas nutiks, jei tinklas pasidalins dėl trikčių?

Yra rinkinys kompromisinių situacijų bandant pasirinkti kaip ir kur saugoti proceso būsenos informaciją ir kada naudoti kešavimą ar replikavimą. Vienu ar kitu atveju tokių scenarijų testavimas tampa sudėtingesnis dėl papildomų sunkumų diegiant skirtingus mechanizmus.

Būkite tikslūs dėl greičio ir patikimumo. Reikia nustatyti, kurios sistemos dalys gali turėti didžiausią įtaką visos sistemos greitaveikai: kur yra „butelio kakleliai“ (*bottleneck*) ir kodėl. Šiuo klausimu gerai gali padėti nedideli testai, skirti išmatuoti sistemos greitaveiką ir nustatyti galimas alternatyvas.

Patvirtinimai (*acks*) yra brangūs ir jų dažniausiai stengiamasi vengti paskirstytose sistemose.

Pakartotini siuntimai (*retransmission*) atima daug laiko. Labai svarbu yra eksperimento būdu nustatyti optimalų vėlinimą, kuris inicijuoja pakartotino siuntimo užklausą.[1][6]

2.2. Paskirstytų sistemų testavimo metodų analizė

Kuriamos sistemos testavimas šiais laikais užima vis daugiau laiko, nepaisant testavimo technologijų tobulėjimo, daugelio testavimo metodikų automatizavimo ir specialių dizaino metodologijų, palengvinančių testavimo procesą, diegimo. Vienos iš pagrindinių to sąlygų yra vis didėjantis sistemų sudėtingumas, sistemų platformų ir techninių komponentų įvairovė bei tendencija apjungti smulkias riboto funkcionalumo programas į didesnes šeimas, paketus.

Taipogi paplitus ir toliau pasaulyje sparčiai plintant internetui ir su juo susijusioms technologijoms vis daugėja šių technologijų panaudojimas verslo vartotojams skirtose programinės įrangos paketuose. Tai sąlygoja spartų paskirstytų sistemų plėtojimąsi. Jau dabar sunku surasti ofisą ar biurą, kuriame nebūtų vidinio organizacijos kompiuterinio tinklo su jame veikiančiais servisais, apimančiais nuo paprasčiausių duomenų mainų tarp vartotojų iki sudėtingos verslo logikos kontrolės sistemų. Todėl plėtojantis paskirstytoms sistemoms turi plėtotis ir tokių sistemų testavimas.

Paskirstytų sistemų testavimas apima daugelį testavimo būdų, kaip ir lokalių sistemų testavimas, tačiau, paskirstytoms sistemoms būdingi specifiniai defektai, kurie yra reti arba visai nepasitaiko lokaliose sistemose:

- *Sinchronizavimo(si) klaidos*: priėjimas prie duomenų privalo būti susinchronizuotas tarp skirtingų komponentų, kitaip neteisingi duomenys gali sukelti neteisingus rezultatus, net jei ir kiekvienas komponentas atliko skaičiavimus teisingai. Ši klaida taip pat būdinga ir lygiagrečioms sistemoms, veikiančioms vienoje mašinoje.
- *Komponentų klaidos*: vienas komponentas sistemoje gali sugesti ir pateikti neteisingus duomenis arba jų visai nepateikti, nors visi kiti komponentai sistemoje veikia nepriekaištingai.
- *Tinklo tarp komponentų klaidos*: tinklas tarp individualių komponentų gali sugesti, nors kiti komponentai gali ir toliau bendrauti tarpusavyje.[13]

Dėl šių specifinių ypatumų testai, skirti testuoti paskirstytas sistemas, privalo atsižvelgti ir apimti galimas specifines problemas, išvardintas aukščiau. Toliau šiame skyrelyje pakalbėsiu apie testavimo metodikas, tinkamas paskirstytų sistemų testavimui, aprašysiu jų privalumus, trūkumus ir taikymus konkrečioje paskirstytos sistemos testavimo situacijoje.

2.2.1. Skaičiuojamieji modeliai (*computational models*)

Lygiagretusis (concurrent) skaičiuojamasis modelis yra būdingas tiek paskirstytoms, tiek lygiagrečioms, tiek lokalioms sistemoms. Jis daro prielaidą, kad daugelis dalykų vyksta vienu metu

(arba beveik vienu metu, bet taip, kad žmogaus akis to nepastebi ir atrodo, kad viskas vyksta vienu metu).

Dažniausiai lygiagretusis skaičiuojamasis modelis sukuriamas pasitelkiant *gijas* – mažiausius savarankiškus vykdymo kontekstus (*context of execution*). Daugelis šiuolaikinių operacinių sistemų leidžia grupuoti proceso gijas į rinkinius, kurie turi bendras savybes bet gali turėti ir privačių duomenų. Tačiau kūrėjas privalo pateikti sinchronizavimo mechanizmą jei skirtingos gijos dirba su bendru duomenų rinkiniu. Objektinės programavimo kalbos palengvina šia situaciją, nes jos pateikia galimybę paslėpti atributus po sąsajom (*interfaces*), o ir atskiros gijos gali būti sukurtos kaip objektai, taip sudarant sąsaja sinchronizavimui. Tačiau standartinės metodikos klasių testavimui negali atrasti sinchronizacijos klaidas, nes objektai turi sąveikauti tarpusavyje, kad atskleisti sinchronizavimo problemas. Tačiau objektinis testavimas leis atrasti ir pašalinti klaidas, nesusijusias su objektų sąveika prieš atliekant sąveikos testavimą, tokiu būdu minimizuojant klaidų kiekį ir jų kilimo galimybę. Lygiagretusis skaičiuojamasis modelis apima sistemas, kuriose kelios gijos veikia viename procesoriuje (vieno ar daugelio branduolių) ir sistemas, kur kelios gijos veikia keliuose fiziniuose procesoriuose, bet dalijasi bendra atmintine.

Tinklinis (networked) skaičiuojamasis modelis pasiekiamas sujungiant atskirus kompiuterius komunikavimo įrangos pagalba. Taip pasiekiamas *fizinis* lygiagretumas, tačiau tokie komponentai tarpusavyje bendrauja lėčiau nei lygiagrečiajame modelyje dėl to, kad komunikacijos įranga yra lėtesnė nei vidinės kompiuterio magistralės. Ir tinkle dirbantys komponentai neturi priėjimo prie bendros atminties (nebent specialiai sukurta), ko pasėkoje turi vykti duomenų mainai per komunikavimo įranga ir specialiai įrengtas sąsajas. Tokiame modelyje sinchronizacijos problema išryškėja labiau dėl to, kad kiekvienas komponentas matuoja laiką nuosavu laiko matavimo įrenginiu (pvz., vietiniu sisteminiu laikrodžiu). Darbas su bendrais duomenimis vykdomas per sąsajas, kurios, jei sukurtos atitinkamai, automatiškai pašalina sinchronizavimo problemas, susijusias su lygiagrečiu priėjimu prie duomenų.

Paskirstytas (pilnai paskirstytas) skaičiuojamasis modelis. Toks modelis palaiko lanksčią architektūrą: sistemoje dalyvaujančių komponentų kiekis gali kisti. Duomenų objektai gali būti paskirstyti kaip tarp atskirų gijų ar procesų toje pačioje mašinoje, taip ir tarp atskirų fizinių kompiuterių. Tačiau tokiose sistemose komponentai turi turėti mechanizmą, padedantį atrasti vienam kitą. Tai gali būti pasiekiama įdiegiant vardų servisą (*naming service*), kuris palaiko pilną komponentų sąrašą ir gali jį pateikti naujai pasijungusiems komponentams (*pvz., DNS servisai internete*). Kita alternatyva yra konfigūracinis failas, kuriame surašyti visų mašinų, galinčių

dalyvauti sistemoje, duomenys. Taipogi ši savybė gali būti standartizuota ir pakartotinai naudojama kitose sistemose be didesnių pakeitimų.[13]

Taigi, kuom paskirstytas modelis skiriasi nuo nepaskirstytų (lokalių) modelių?

Nedeterministiškumas (nondeterminism)

Kadangi kodo vykdymas gali priklausyti nuo išorinių duomenų (pvz., atvykstančių pranešimų eiliškumas serverio procese), tai testavimas gali ne visada gauti tokius pačius rezultatus, net jei visi skaičiavimai buvo atlikti nepriekaištingai. Priklausomybė yra ir nuo operacinės sistemos planavimo mechanizmo (*scheduler*), nes gijos gali būti vykdomos ne ta pačia tvarka kiekvieną kartą. Ne gana to, sistemos testavimo rezultatus gali įtakoti ir pokyčiai kituose, su testuojamąja sistema tiesiogiai nesusijusiais, komponentais (pavyzdžiui, gali paveikti testuojamosios sistemos gijų vykdymo seką). Svarbiausia, jei defektas nepasikartoja, tai dar nereiškia, jog jis yra visiškai ištaisytas.

Nedeterministiškumo įtaką testavimo metu galima sumažinti sudarant standartizuotą testavimo aplinką sistemos komponentams:

- „Švari“ mašina, su minimaliu kiekiu bendrai naudojamų įrenginių.
- Griežtai nustatytas programų, kurios būtinos platformos funkcionavimui, kiekis.
- Bendros programos, kurios gali būti tipinėje mašinoje, kurioje veiks sistemos komponentai.
- Kiekvienas testavimo atvejis apibrėžia, kokios modifikacijos atliekamos šiai standartinei testavimo aplinkai, įskaitant ir apibrėžtą procesų startavimo eiliškumą.
- Derinimo įrankis turėtų būti įtrauktas į standartinę testavimo aplinką.
- Bent jau pradžioje reikėtų atskirti testavimo mašiną(-as) nuo tinklo.[13]

Nedeterministiškumas paskirstytose sistemose yra viena iš pagrindinių *heisenbug* tipo klaidų atsiradimo priežastis, ir kartu tai yra pagrindinė tokių sistemų testavimą ir derinimą apsunkinanti savybė.

Papildoma infrastruktūra

Dauguma paskirstytos sistemos komponentų pasikliauja trečiųjų šalių kurta infrastruktūra (kompiuterių platformos, operacinės sistemos, tinklo įranga ir t.t.). Todėl reikėtų sukurti *regresijos* (*regression*) testų rinkinį, skirtą patikrinti suderinamumą tarp programos ir infrastruktūros. Šie testai turėtų būti naudojami tikrinant naujas infrastruktūros versijas (ar naujus komponentus), ar tiesiog rekonfigūruojant sistemą.

Kai kurios infrastruktūros yra savaime besikeičiančios (*self-modifying*) ir rekonfigūruojasi kai sistema persikonfigūruoja, taip sukeldamos papildomų problemų sistemos kūrėjams. Tokiu atveju specifinis duomenų rinkinys gali sukelti kitą vykdymo kelią, nes senasis jau nebeegzistuoja dėl infrastruktūros pokyčių.

Daliniai sistemos trikčiai (partial failures)

Šie trikčiai kyla kai dėl programinių ar techninių gedimų mašina, kurioje patalpinti sistemos komponentai, negali veikti. Šio tipo gedimus galima *simuliuoti* testavimo metu tiesiog atjungiant tinklo jungtis ar atjungiant kitus susijusius su testuojamuoju tinklo ar sistemos komponentus.

Neatsakomumas (time-out)

Tinkle veikiančios sistemos išvengia aklaviečių (*deadlock*) nustatydamos tam tikro neatsakomumo laikotarpį visoms užklausoms į kitus sistemos komponentus. Jei per nustatytą laiko tarpą negaunama jokio atsakymo užklausa atmetama. Todėl sistema turi elgtis atitinkamai, kai užklausa atsakoma ir kai neatsakoma (nors, ko gero, skirtingai). Testavimas turėtų įtraukti skirtingas apkrovų konfigūracijas mašinoms ir tinklui bei *tyčia* sukelti dalinius trikčius, kad užtikrinti, jog sistema veiks tinkamai nutikus neatsakymams realioje darbo aplinkoje.

Sistemos dinamiškumas (dynamic nature)

Paskirstytos sistemos neretai sukuriamos išlaikant galimybę pakeisti jų konfigūraciją. Pavyzdžiui, specifinės užklausos gali būti dinamiškai paskirstomos priklausomai nuo atskirų mašinų apkrovos. Taipogi sistemoje gali dalyvauti kintamas mašinų (kartu ir komponentų) skaičius. Todėl testai turi atspindėti ir šią sistemos savybę ir vykdomi su skirtingomis galimomis konfigūracijomis.

Visos šios savybės tiesiogiai įtakoja paskirstytų sistemų testavimą.[13]

2.2.2. Sinchronizacija: metodikos ir testavimas

Sinchronizacija sistemoje yra privaloma, kai daugiau nei vienas objektas gali norėti pasiekti tuos pačius duomenis vienu metu. Ypač atidžiai reikia elgtis, kai šie procesai nori modifikuoti duomenis.

```
public double calculateSomething()
{
    int a = memory.GetInt(0x40242434);
    int b = memory.GetInt(0x43572473);
    int c = a+b;
    memory.PutInt(0x40242434, c);
    double d = memory.GetDouble(0x45675767);
    return c*sin(d);
}
```

4 pav. *Sinchronizuojamos gijos kodo fragmentas.*

Apžvelkime kodo fragmentą dešinėje. Jame yra aprašytas labai paprastas metodas, kuris pasiima tris reikšmes iš atminties (*memory* objekto *GetInt()* ir *GetDouble()* metodai, kuriems per parametrus

nurodomas konkretus atminties adresas, iš kurios imti duomenis), pakeistų reikšmių įrašo atgal į atmintį (ten pat, iš kur paimta objekto *a* reikšmė). Pavyzdžiui, taip gali būti naudojama bendra kažkokių duomenų indekso (ar duomenų srauto pozicijos rodyklės) reikšmė. Jei šis kodas vykdomas vienoje gijoje, tai problemų nekyla. Tačiau jei vienu metu tą patį metodą bando įvykdyti daugiau nei viena gija, gali kilti nesklandumų. Jei *memory* objektas neturi metodų užklausų sinchronizavimui, gijos gali sutrukdyti viena kitai teisingai atlikti užduotį – viena gija gali pakeisti duomenis kitai gijai dar nebaigus skaičiavimo. Geriausiu atveju gali būti gauti tie patys rezultatai du sykius ir nieko blogo nenutiks. Tačiau dėl tokios menkos problemos gali sutrikti visos sistemos darbas, jei nuo šitos funkcijos priklauso kokios nors labai svarbios procedūros darbas.

Populiariausios metodikos sinchronizavimui valdyti:

- Barjeras (*barrier*): Kai grupei gijų ar procesų nustatomas taškas, ties kuriuo gija/procesas privalo sustoti ir negali tęsti vykdymo kol šio taško nepasieks visos gijos/procesai.
- Rakinimas (*lock*) ar semaforai (*semaphore*): sinchronizacijos metodas, kai viena gija/procesas nustato semaforą (ar užrakina priėjima prie tam tikrų duomenų) kitoms gijoms, kurios turi laukti, kol semaforas/užraktas bus nuimtas, kad toliau tęsti apdorojimą šių duomenų. Šis metodas gali sukelti aklavietės (*deadlock*) situaciją.
- Gijų jungimas (*thread join*): kai kelios gijos (laikinai) apjungiamos į vieną, kad atlikti jom bendrą operaciją.
- Muteksas (*mutex*): rakinimo ir semaforu atmaina, kai gijos ar procesai nustato tam tikrą objektą, kuris reguliuoja gijų/procesų priėjimą prie svarbių resursų.
- Seniūnai (*monitors*): specialus objektas, pasikliaujantis muteksus ar rakinimo metodikas, kad sinchronizuoti gijų darbą. Skirtumas tame, kad tik vienas seniūno objekto metodas gali būti vykdomas bet kurios gijos vienu metu, taip išvengiant konfliktų tarp gijų.
- Įvykiai (*events*): speciali sinchronizacijos metodika, kai laukiančiajai gijai/procesui pranešama, kad specifikuota sąlyga įvyko.[14]

Bet kokiu atveju, sinchronizacija privalo būti aprašyta sistemos kūrimo metu. *Kelias* yra rinkinys logiškai susietų sakinių, kuris yra vykdomas kai įvedama tam tikra informacija. Svarbūs keliai paskirstytose sistemose:

- **SYN-įvykis (*SYN-event*)**: bet koks veiksmas, sukeliantis gijų sinchronizavimą. Naujos gijos sukūrimas yra vienas iš daugelio pavyzdžių. Kiti SYn-įvykių pavyzdžiai yra gijos sunaikinimas, gijos objekto sunaikinimas, pranešimo persiuntimas iš vieno gijos objekto į kitą ar tiesiog gijos būsenos (pristabdymas ar paleidimas) valdymas iš kitos gijos ar gijos objekto.

- **SYN-seka** (***SYN-sequence***): SYN-įvykių seka, kuri visada įvyksta tam tikra tvarka.

Testuojant testavimo atvejai turėtų būti kuriami atsižvelgiant į SYN-sekas. Pagrindinė gija neturėtų būti įtraukiama į galimų kelių skaičių, nes ji yra vykdoma visada, nepriklausomai nuo nustatytų duomenų. Testuotojas seka kelią tarp vieno SYN-įvykio prie kito.

Bet koks objektas, turintis nuosavą giją, turi būti nuodugniai patikrintas atkirai, kaip klasė, ir tik tada testuojamas sąveikoje su kitais objektais.

SYN-kelio technika skirta atrasti defektams, kylantiems dėl sinchronizacijos problemų. Tačiau ši technika nepakeičia įprastų testavimo technikų, skirtų atrasti kitokio tipo defektus, nesusijusius su sinchronizacija.

2.2.3. Kūrimas testuojamumui (*design for testability*)

Labai svarbu kurti tokį kodą, kurį po to būtų lengva testuoti. Tam buvo sukurtas kūrimo testuojamumui metodas.

Šią kūrimo metodiką pirmi pradėjo naudoti elektronikos inžinieriai, siekdami palengvinti jų funkcijų testavimą pasitelkiant automatinę testavimo įrangą (*ATĮ*). Tam specialūs testavimo taškai buvo iškeliami į plokštės kraštus, kad palengvinti priėjimą prie jų. Palengvindami testavimo įrangos priėjimą, kūrėjai galėjo ištestuoti daugiau plokštės funkcijų, taip sumažindami gedimų skaičių. Ta pati technologija gali būti pasitelkiama ir techninės įrangos palaikymui pas užsakovą, bei gedimų diagnozavimui.

Technologija, jau seniai naudojama elektronikos inžinerijoje, taip pat sėkmingai gali būti pritaikyta ir programinės įrangos inžinerijoje.

James Bach apibrėžia testuojamumą, kaip sistemos savybių, palengvinančių programų testų kūrimo bei vykdymo procesą. James Bach taipogi apibrėžia kelis programos testuojamumo terminus:

- **Kontrolė**. Kuo geriau mes galim kontroliuoti programinę įrangą, tuo lengviau mes galime automatizuoti jos testavimo procesą.
- **Matomumas** (***visibility***). Ištestuoti galime tik tai, ką matome.
- **Funkcionalumas** (***operability***). Kuo geriau veikia programa, tuo efektyviau ją galima ištestuoti.
- **Paprastumas**. Kuo mažiau reikia testuoti, tuo greičiau ir patikimiau galima ištestuoti.
- **Suprantamumas**. Kuo geriau aprašyta programa, tuo geriau ji yra suprantama, tuo protingiau ją galima ištestuoti.

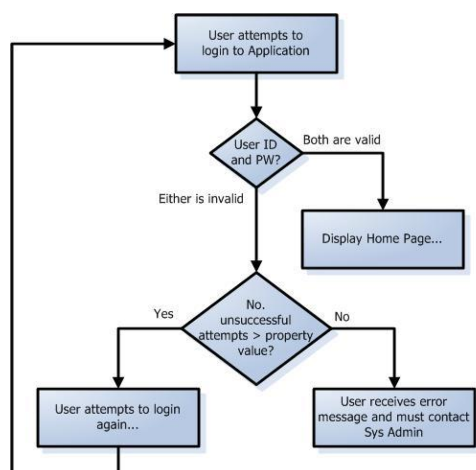
- **Tinkamumas.** Kuo daugiau mes žinome ir suprantame apie programos paskirtį, tuo geriau galime suorganizuoti testavimo procesą svarbių klaidų paieškai.
- **Stabilumas**. Kuo mažiau pakeitimų, tuo mažiau kyla trukdžių testavimui.[16]

Testuojamumas yra dizaino problema ir į jį reikia atsižvelgti dar kuriant visos sistemos dizainą. Projektai, kurie atideda testavimą vėlesniems projekto kūrimo etapams neretai susiduria su sunkumais modifikuojant programų kodą, kai testuotojai imasi darbo ir pradeda siūlyti pakeitimus. Testuojamumui reikia kooperacijos, įvertinimo ir komandos įsipareigojimo programinės įrangos patikimumui.[15]

Prastai surašyti reikalavimai gali sukelti problemų tiek kūrėjams, tiek testuotojams. Tokius reikalavimus yra ne tik sunku suprasti, bet ir neretai jie gali būti suprantami dvejopai ar net daugiau. Neretai reikalavimų perrašymas, kad jie būtų suprantamesni, neatliekamas dėl kelių priežasčių: laiko trūkumas, autorius nemato prasmės ir t.t. Taigi, testuotojui telieka spėlioti, kaip kūrėjas interpretavo reikalavimą. Siekiant paaiškinimų iš reikalavimų autoriaus neretai kyla papildomų nusivylimų, o klausti kūrėjo, kaip jis interpretavo reikalavimą dažniausiai sukelia daugiau klausimų nei paaiškina.

Kartais testuotojams tenka perrašyti reikalavimus tokiu būdu, kad jie būtų vienareikšmiškai ir lengvai suprantami, ir tada pateikti juos kūrėjui/užsakovui, kad tie patvirtintų, jog jie tą ir norėjo išreikšti. [15]

Todėl gerai aprašyti reikalavimai sistemai yra taip pat svarbūs, kaip ir tinkamai parašytas programos kodas, o gal net ir svarbesnis. Yra atskira programų inžinerijos šaka, studijuojanti, kaip tinkamai aprašyti programų reikalavimus: reikalavimų programinei įrangai inžinerija.



5 pav. ***Veiksmų diagramos reikalavime programai pavyzdys.***

Yra kelios kūrimo testuojamumo metodikos, skirtos palengvinti reikalavimų suprantamumą ir sumažinti problemas, kylančias dėl neteisingai suprastų reikalavimų. Svarbiausia, reikia vengti kalbos konstrukcijų, kurios gali būti suprastos dvejopai. Be to, gera, rišli kalba pagerina bendrą reikalavimo teksto suprantamumą. Trumpi sakiniai geriau išreiškia užduotį už ilgus, sudėtingus. Geriausia praktika yra stengtis laikytis 30 žodžių sakiniui „ribos“ ir stengtis naudoti

esamąjį laiką bei nemaišyti laikų sakinyje. Rašytojo patyrimas kuriant programinę įranga atlieka nemažą vaidmenį gerų reikalavimų rašymo procese. Dažniausiai verta sudaryti techninių terminų žodyną, ypač jei ties reikalavimų specifikavimu dirba skirtingų profesijų žmonės. Ir svarbiausia, rašant reikalavimus neapsiriboti vien natūralia kalba, būtina pasitelkti papildomas priemones: įvairiausias diagramas: panaudos atvejų, darbo eigos, teisingumo lenteles, sintetines kalbas sudėtingesnėms algoritmo vietoms aprašyti ir pan. [15]

Kodo rašymo eigoje reiktų pasistengti, kad visos klasės (objektai), o ypač tos, kurios vykdo bendravimą tarp komponentų, turėtų griežtai apibrėžtas sąsajas (*interfaces*). Vienas iš patogiausių būdų objektiniame programavime yra kiekvienai specifinei klasei sudaryti sąsajos klasė, iš kurios paveldėti sąsajos metodus jau naudojamai klasei. Iš to paties sąsajos objekto paveldėta testavimo klasė palengvins testavimą, taipogi sąsajos gali būti taikomos ir kitokiose testavimo technikose, tokiose, kaip, pavyzdžiui, „kamščių ir valdiklių“ testavime. Kiekviena klasė turėtų turėti *konstruktorių pagal nutylėjimą (default constructor)* – konstruktorių, kuris inicializuoja visas komponento/klasės vidines reikšmes reikšmėmis pagal nutylėjimą, taip galima išvengti sudėtingos komponentų hierarchijos testavimo metu.[13]

2.2.4. Pagrindinės testavimo metodologijos

Paskirstytos sistemos turi nemažai unikalių savybių, kurių neturi lokalios arba lygiagrečios sistemos, tačiau jų testavimui naudojamos ir kitos metodikos, bendros visoms programinės įrangos rūšims.

Statinis testavimas

Statinis testavimas yra programų testavimas realiai jų kodo nevykdant. Toks testavimas dažniausiai atliekamas iškarto programos kūrimo aplinkoje (*integrated development environment*) pasitelkiant specialias komandas, to naujausiose aplinkose šis procesas neretai vyksta lygiagrečiai su programos kodo rašymu, fone.

Statinis testavimas apima kodo analizę, kai programinė įranga ar žmogus peržiūri kodą tam, kad surasti galimas klaidas, nesaugias programavimo metodikas ar tiesiog statistiškai daugiausiai problemų sukeliančias kodo ar programavimo kalbos konstrukcijas. Modernios programavimo aplinkos (pavyzdžiui, Microsoft Visual Studio) atlieką ši tikrinimą fone, programuotojui dar tik rašant kodą, ir automatiškai pasiūlo būdus, kaip pataisyti galimas problemas. Taipogi dauguma šiuolaikinių kompiliatorių gali atlikti ribotą statinę kodo analizę kompiliavimo metu.

Vienetų (unit) testavimas

Smulkiausių kodo vienetų testavimo metodika. Kodo vienetu laikoma savarankiška procedūra ar metodas, turinti griežtai apibrėžtą reikšmių sritį ir duodanti rezultatus iš apibrėžtos rezultatų srities. Kadangi dirbama su smulkiais savarankiškais kodo vienetais, ši testavimo metodika šiuo metu yra beveik pilnai automatizuota. Daugelis modernių programavimo aplinkų pateikia įrankius vienetų testavimui, tačiau yra ir nemažai savarankiškų vienetų testavimo įrankių skirtingoms programavimo kalboms.

Vienetų testavimas ieško problemų smulkiausiuose programos komponentuose, kviečiant testuojamas funkcijas su iš anksto nustatytais parametrais ir tikrinant, ar rezultatai atitinka iš anksto paskaičiuotus. Vienetų testavimo metu sukurti testai gali būti pakartotinai panaudojami visose programos kūrimo stadijose.

Tačiau vienetų testavimas remiasi programų algoritmų *determinizmu* – t.y. savybe, kad kiekvieną syki paleistas metodas ar procedūra su tais pačiais duomenimis gražins tuos pačius rezultatus. Tačiau paskirstytose sistemose neretai vyrauja *nedeterminizmas*, ko rezultate vienetų testavimas turi gan ribotą panaudojimą sistemos vidinėms, deterministinėms, funkcijoms testuoti.

Objektinis testavimas

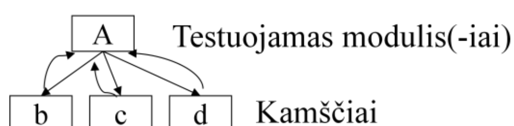
Objektinio testavimo paskirtis yra testuoti mažiausius savarankiškus sistemos komponentus – objektus. Objektą dažniausiai sudaro viena ar kelios klasės, kurios pateikia griežtai apibrėžtą sąsają darbui su jais ir turinti kelis vidinius, iš išorės nepasiekiamus kintamuosius.

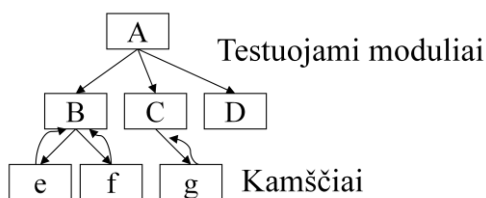
Objektinio testavimo yra atliekami visi įmanomi veiksmai su objektu: sukuriami klasės objektai, nustatomos visi objekto atributai, iškviečiami visi objekto metodai ir stebimas objekto veikimas. Objektinis testavimas gali būti *juodos dėžės (black box)* ir *baltos dėžės (white box)*. Juodos dėžės testavimo metu objektas traktuojamas kaip juoda dėžė, su tam tikru savybių rinkiniu, pasiekiamu iš išorės, tačiau vidinis objekto veikimas nėra žinomas. Tikrinama, ar visi metodai gražina tokius rezultatus, kokių tikimasi. Baltos dėžės testavimas traktuoja objektą kaip atvirą, baltą, dėžę. Testavimo metu kviečiami metodai ir stebimas vidinis objekto veikimas – kokie kintamieji keičiasi, kaip sąveikauja metodai, ar padengiamas visas kodas ir pan. Klasių paveldimumas gali sukelti problemų testuojant baltos dėžės principu, nes objekto veikimas, kintamieji ir metodai būna paskirstyti po kelias vietas, klases. Objektinius testus stengiamasi sudaryti taip, kad padengti maksimalų objekto kodo dalį, patikrinti visas galimas būsenas ir perėjimus.

Kadangi vienas objektas dažnai realizuoja kurį nors vieną ar kelis tarpusavyje susietus sistemos panaudojimo atvejus, testavimo metu neretai remiamasi projekto dokumentacija ir, specifiškai, panaudos atvejų specifikacija. Panaudos atvejų testavimas daugiau apima juodos dėžės testavimą, nes baltos dėžės testavimo metu būtina remtis ir testuojamojo komponento kodu.

Kamščiais (stubs) ir valdikliais (drivers) pagrįstas testavimas

Neretai sistemai didėjant, ypač jei ją kuria daugiau žmonių ar komandų, tenka pratestuoti tik tam tikrą kodo dalį (pavyzdžiui, atskira paskirstytos sistemos komponentą). Tam, kad nereikėtų surinkti ir paleisti visos sistemos (o neretai visa sistema net nėra dar pasiekiama testavimui), yra naudojami kamščiai ir valdikliai. Kamščiai ir valdikliai – tai pagalbiniai moduliai, kurie simuliuoja atitinkamų realių komponentų veikimą.

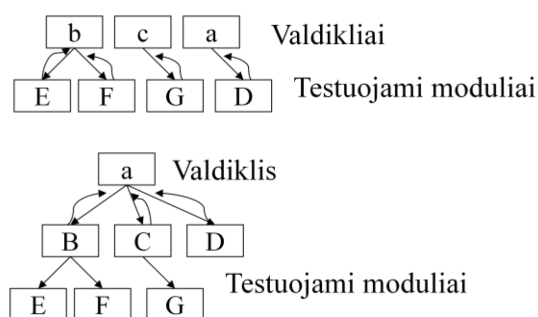




6 pav. ***Kamščių panaudojimo pavyzdys.***

Kamščiai naudojami pakeisti žemesnio lygio komponentus jų abstrakcijomis. Pavyzdžiui, komponentas kreipiasi į kitą komponentą, atsakingą už duomenų gavimą iš kokio nors duomenų šaltinio (interneto, failo, duomenų bazės, serverio ar pan.). Tokį komponentą pavaduojantis kamštis tiesiog gražintu jau iš anksto nurodytą duomenų rinkinį, su kuriuo ir dirbtų testuojamas modulis. Tačiau kamštis neatlieka jokių modulio, kurį jis pakeičia, apart būtiniausiu testuojamajam moduliui, funkcijų.

Valdiklis yra programinė priešingybė kamščiui, tačiau jie tarpusavyje tampriai susiję. Valdiklis skirtas patikrinti komponento veikimą „iš viršaus“, t.y. valdiklis naudojasi komponento, kurį testuojama, paslaugomis. Valdiklis gali, pavyzdžiui, iškviesti duomenų apdorojimo metodus testuojamame objekte ir sulyginti rezultatus su iš anksto nurodytais.



7 pav. ***Valdiklių panaudojimo pavyzdys.***

Tiek kamščiai, tiek valdikliai taipogi gali atlikti duomenų, kuriuos perduoda testuojamasis komponentas, patikrinimui, ar sąsajos su komponentu testavimui. Paskirstytose sistemose kamščiai ir valdikliai plačiai naudojami testuoti atskirų sistemos komponentų veikimą, sqsajos tarp jų testavimui ir patikrinti, kaip komponentas elgiasi nestandartinėmis situacijomis (pvz., dėl ryšio

trikties buvo gauti ne visi duomenys, ar komponentas atliko užklausą, bet serveris neatsakė per paskirtą laiką (*time-out*).

Valdikliai ir kamščiai taipogi gali būti naudojami sąsajoms tarp komponentų specifikuoti ir testuoti, kai dar neturima pačių komponentų – pavyzdžiui, kūrimo pradžioje. Vėliau pasitelkiant valdiklio ir kamščio kodu yra kuriami pagrindiniai komponentai, naudojantys tą sąsaja.

Kai nepaskirstytose sistemose testavimo metu galima išsiversti be kamščių ir valdiklių, paskirstytose sistemose, pagrinde dėl jų dydžio ir tarpusavio sąveikos tarp komponentų sudėtingumo, be šios metodikos beveik neįmanoma tinkamai ištestuoti sistemą. Be to, kamštis (ar valdiklis) gali būti parašyti kaip automatizuoti komponentų testai.[13]

Apračiau tik pačias pagrindines testavimo metodikas. Yra begalė skirtingų testavimo metodikų, kurias galima pritaikyti tiek paskirstytom, tiek lokaliom sistemoms. Be to, dažniausiai vienu metu taikomos kelios skirtingos testavimo metodikos. Kai kurios metodikos reikalauja atitinkamos programos kūrimo stadijos (pavyzdžiui, neįmanoma atlikti klasių testavimo, jei dar nepradėtas rašyti kodas).

3. Veidą atpažįstančios įeigos kontrolės sistemos analizė

3.1. Panašios sistemos

Panagrinėjus rinką, nustatyta, kad panašių sistemų nėra daug. Toliau palyginamos kelios sistemos.

3.1.1. IITS FRS ACCESS

IITS FRS ACCESS yra moderni technologija, apjungianti saugumą ir biometrinę prieigą per elektroniniu būdu valdomas duris. Sistemos veikimo principas:

- Vykdomas skenavimas, norint aptikti, kad asmuo yra prie kameros.
- Atliekama paieška duomenų bazėje, norint aptikti, ar įrašas apie asmenį jau egzistuoja.

8 pav. ***ITS FRS Access sistema.***

- Identifikavus vartotoją, sistema automatiškai atidaro duris ir išsaugo informaciją į duomenų bazę.
- Produkto savybės:
- Greitas ir tikslus veido aptikimas
- Pilnas nuotolinis valdymas per tinklą
- Šiuolaikinis dizainas
- Žurnalizavimas
- Kalbėjimas vietine kalba
- Paruošta sujungti į tinklą

Naujas vartotojas užfiksuojamas jam natūralia išraiška žiūrint į kamerą. Tada jo veido savybės išsaugomos duomenų bazėje. Ši informacija naudojama vėliau, norint palyginti į patalpą bandančio patekti žmogaus veido atvaizdą su esančiais duomenų bazėje. Jei randamas atitikmuo, sistema automatiškai atidaro duris ir vartotojui leidžia įeiti/išeiti.

Kiekvieną kartą vartotojui bandant įeiti/išeiti, sistema įrašo datą, laiką ir duomenis apie vartotoją tam, kad būtų galima gauti pilną informaciją iš kiekvienos dienos.[19]

3.1.2. „NeoFace Control“

„NeoFace Control“ – pažangi technologija, naudojanti biometrines žmogaus savybes, skirta žmogaus tapatybei nustatyti. Vietoj to, kad būtų naudojamos tradicinės įeigos kontrolės metodai, tokios kaip slaptažodis ar identifikavimo kortelės, „NeoFace Control“ taiko veidą atpažįstančią įeigos kontrolę. Tai yra higieniška ir nereikalauja jokio fizinio kontakto su įranga.

9 pav. ***"NeoFace Control" priekinis skydelis.***

Kai asmuo stovi priešais į sieną įmontuotą įrangą su vaizdo kamera, tai leidžia lengviau nustatyti žmogaus veido kontūrus. Ši sistema automatiškai užfiksuoja veido atvaizdą ir mažiau nei per vieną sekundę palygina jį su paruoštų veidų duomenų baze. Jeigu duomenys sutampa, asmuo bus įleistas, kitu atveju įėjimas bus draudžiamas. Vartotojas gali pasirinkti, kad būtų parodomas pranešimas su jo vardu ir laikas, kada ir kur jis lankėsi.

„NeoFace Control“ sistema yra mažiausia bei patikimiausia veidą atpažįstanti įeigos kontrolės sistema. Turi draugišką vartotojui sąsaja, aukštą saugumo lygį, žurnalizavimo funkcijas.

Žmogaus atpažinimas nenaudojant jo tiesioginio prisilietimo prie atpažįstančio įtaiso. Norint patekti į patalpas būtinas veido atpažinimas. Tai atlikti padeda terminalas, kuris nereikalauja jokio fizinio kontakto. Atėjusio žmogaus veidas yra užfiksuojamas vaizdo kamera ir palyginamas su jau duomenų bazėje esančiais atvaizdais. Tai leidžia sumažinti rizikos faktorius, tokius kaip slaptažodžio užmiršimas, vagystės ar įeigos duomenų dalinimasis, norint išlaikyti aukštą saugumo lygį.

Sumažina kompanijos išlaidas, be to saugo gamtos resursus. „NeoFace Control“ sistemą galima integruoti į daugelį kitų sistemų, tokias kaip vaizdo stebėjimo sistemas. Naudojant šią sistemą nebereikia gaminti identifikavimo kortelių, dėl to mažinamas popieriaus suvartojimas kompanijoje.

Atpažinimo algoritmas. Sistema fiksuoja veido atvaizdą bei palygina jį su duomenų bazėje esančiais atvaizdais. Nustatymo algoritmas pagerina greitaveiką, tikslumą apdorojant, lyginant atvaizdus. Tai leidžia identifikacijos proceso laiką sumažinti iki 1 sekundės.

Įsilaužimo aptikimas bei sekimas realiu laiku. Ši įeigos kontrolės sistema geba išsaugoti foto bei žurnalizavimo įrašus vartotojų, kurie nebuvo įleisti. Taip pat suteikia galimybę stebėti darbuotojų judėjimus realiu laiku, užtikrina tikslumą ir išvengia įsilaužimo galimybės.

Rinkos ir kvalifikacija. Šią sistemą naudoja valstybinės bei privačios įmonės 9 šalyse. Teigiami atsiliepimai įrodo, kad sistema sustiprina verslo bei saugumo efektyvumą. „NeoFace Control“ įgijo tarptautinius sertifikatus plėtojant saugumą bei gaminant technologijas, kurios padeda saugoti gamtos resursus.

Komponentai, sudarantys „NeoFace Control“ sistemą

Veido atpažinimo skydelis: Į sieną įmontuota kamera su LCD ekranu, skirtu pranešimams rodyti.

Procesorius: Programinė įranga skirta veido nustatymui, administratoriaus valdymo skydelis bei registravimo sistema.

Administruojamos sritis: vartotojo registravimas, darbuotojų žurnalizavimas naudojant TCP/IP tinklą.[20]

3.1.3. FxGuard Pro

10 pav. *"FxGuard Pro" skydelis.*

Biometrinė prieigos kontrolės sistema, naudojanti veido atpažinimo technologiją.

Savybės:

- Sistema veikia daugiausia TCP/IP pagrindu.
- Lengva integruoti į korporacijos LAN.
- Įrodytas greitis ir tikslumas.
- Suderinamumas keliose platformose.
- Įmontuotas infraraudonųjų spindulių jutiklis ir RFID skaitytuvas.
- Veido atpažinimas užtrunka mažiau nei 1 sekundę.

Technologija yra draugiška vartotojui, o veido atpažinimas užtrunka mažiau nei vieną sekundę. Norint sustiprinti saugumą, „FxGuard Pro" taiko dvigubą patikrinimą. Sistema yra tinkama naudoti viešose parodose bei verslo ar valstybinėse įmonėse.

„FxGuard Pro" galima lengvai integruoti į vietinį tinklą (LAN) ir darbo stotį ar serverį. Sistemos komponentai daugiausiai yra sujungti per TCP/IP protokolą. Tai užtikrina suderinamumą tarp platformų, o vartotojams nereikia įdiegti specialios programinės įrangos visuose kompiuteriuose.

Visa sistemos programinė įranga yra įmontuota į įrenginį, todėl nereikia naudoti atskiro kompiuterio. Taip sumažinami sistemos kaštai.[21]

3.2. Panašių sistemų palyginimas

1 lentelė. ***Veidą atpažįstančių įeigos kontrolės sistemų palyginimas.***

Kompanijos pavadinimas	Produkto pavadinimas	Produkto aprašymas	Produkto ypatybės
Ex-Sight	„FRS – OnTime“	Vykdomas automatinis žmogaus veido nuskenavimas. Pradedama paieška duomenų bazėje. Jeigu atitikmuo buvo rastas, sistema atidaro duris ir išsaugo informacija duomenų bazėje. Sistemą galima valdyti nuotoliniu būdu. Visi nustatymai gali būti programuojami internetu.	Atpažinimo greitis: 100,000 veidų per sekundę Vieno kadro apdorojimas Laikas: 0.15 s. Registracijos laikas <= 1s. Identifikacijos metodas: 1:1, 1:N Veido šablono dydis: 2.3 KB veidui Žurnalizavimas: iki 1,000,000 įrašų Bazinė licencija: 200 vartotojų
NEC	„NeoFace Control“	Tai šiuolaikinė įėjimo kontrolės sistema atpažįstanti žmogaus veidą. Veikimo principas: veido išskyrimas->veido paieška duomenų bazėje->jeigu atitikmuo rastas, atidaromos durys. Ši sistema pasižymi draugiška vartotojo sąsaja, labai kompaktišku bei patogiu įdiegimu. Lengvai integruojama į jau veikiančias įvairias stebėjimo sistemas.	Bazinėje versijoje galima naudoti iki 500 darbuotojų veidų ruošinių. Verifikacijos laikas - 1s. Identifikacijos metodas - 1:N arba 1:1. Monitorius - 2.5” spalvotas vaizdo ekranas Kameros tipas - CCD su infraraudonaisiais spinduliais. Operacinė sistema - Windows XP Reikalavimai kompiuteriui - P4 3.0GHz, 512 MB RAM, 40GB HDD, TCP/IP, WXXP ir vaizdo įrašymo plokštė

RCG	„FxGuard Pro Biometric Access Control with Face Recognition Technology“	Ši sistema siūlo tikslų bei efektyvų žmogaus veido atpažinimą. Ji lengvai ir greitai įdiegiama. Pasižymi išplečiamumu bei draugiška vartotojo sąsaja. Šios sistemos į kompiuterį įdiegti nereikia, kadangi visas valdymas atliekamas internetu TCP/IP protokolu. Tai užtikrina darbą skirtingomis OS.	Monitorius - 3.5” TFT 262K spalvų LCD Kameros tipas - CCD su infraraudonaisiais spinduliais Verifikacijos laikas – mažiau nei 1s. Neteisingi atmetimai – mažiau nei 2% Neteisingi atpažinimai – mažiau nei 0.01% Protokolas - TCP/IP Naudojama atmintis - 512MB Compact Flash, iki 17,000 įrašų

Apžvelgus keletą sistemų galima daryti išvadą, kad visų jų atliekama užduotis yra ta pati: išskirti atėjusio žmogaus veidą, palyginti jį su duomenų bazėje esančiais veidų aibe, jeigu atvaizdas atpažįstamas – durys atidaromos. Skirtumai tarp šių sistemų: naudojama aparatūra, protokolai, algoritmai. Nuo šių dalių suderinimo ir priklauso, kaip greitai ir efektyviai vyks procesas.

4. Veidą atpažįstančios įeigos kontrolės sistemos projektavimo ypatumai

Kiekvienos sistemos kūrimo pradžioje reikia ištirti rinką, peržiūrėti galimus konkuruojančius produktus, aptarti, kokios technologijos ir/ar programiniai produktai bus naudojami sistemoje ir sistemos kūrimo metu, sudaryti detalų veikimo planą, surinkti reikalavimus. Labai projektavimo procese padeda modeliavimas, nes pagal sukurtą modelį galima lengviau projektuoti sistemą bei ją suprojektavus atlikti testavimą. Modelis – kaip atskaitos taškas sistemos kūrime.

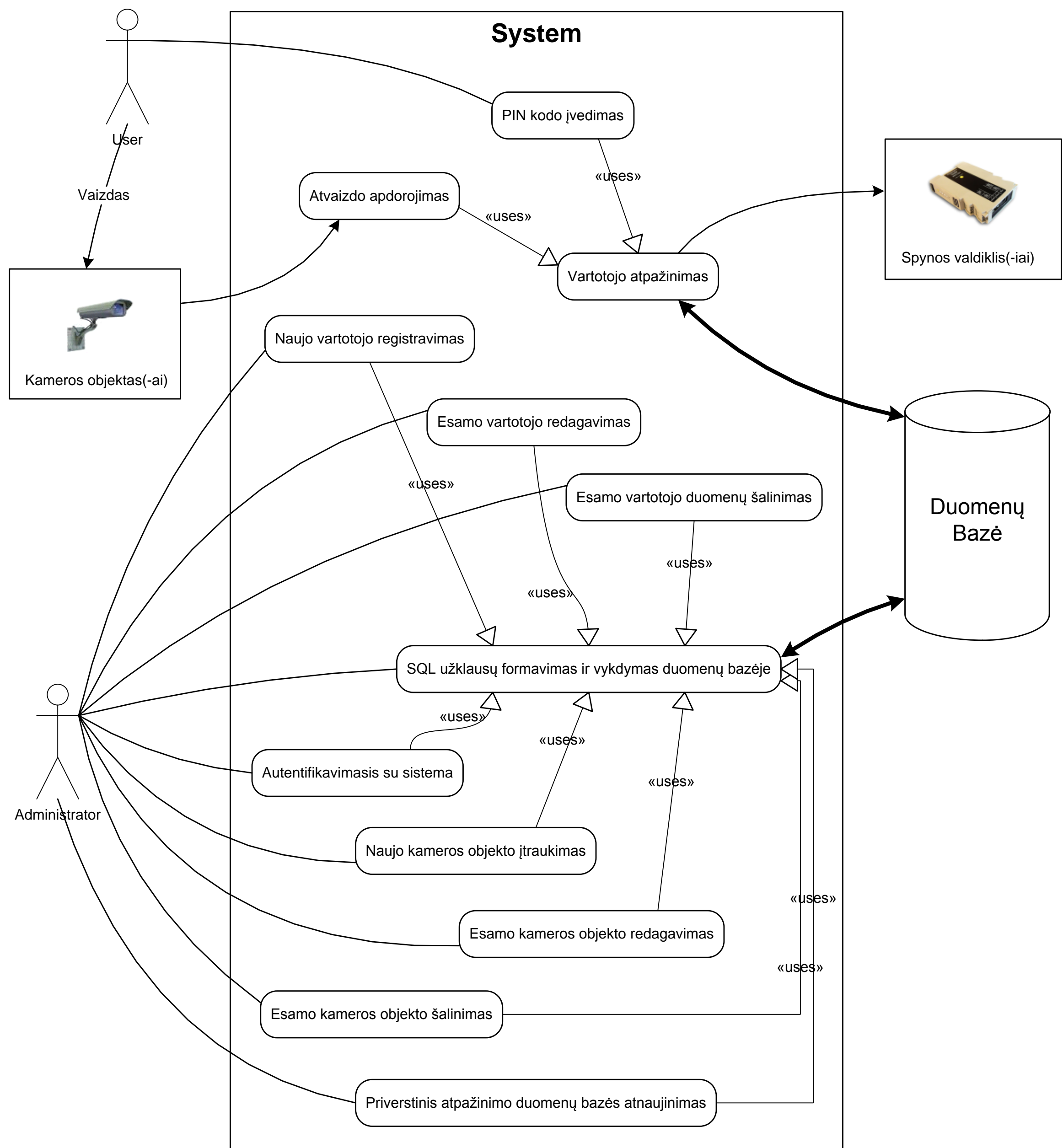
Šiame skyriuje aptarsiu sistemos architektūrą, jos tikslus ir apribojimus, naudojamas technologijas, sistemos architektūros sudedamas dalis (komponentus) ir detaliai aprašysiu kiekvieno komponento ypatumus sistemoje.

4.1. Architektūros tikslai ir apribojimai

1. COTS produktų panaudojimas:
    a. Naudojama *OpenCV* atvirojo kodo biblioteka darbui su vaizdo gavimo ir apdorojimo technine įranga, todėl reikia parinkti kameras, suderinamas su šiom bibliotekom;
    b. Programinė įranga kuriama Windows operacinių sistemų šeimai (*XP, 2003 Server, Vista, 2008 Server, 7, 2008 Server R2* ir naujesnėm) todėl netinka sprendimai, skirti kitoms OS šeimoms;
    c. Kadangi visi sistemos komponentai tarpusavyje bendrauja *TCP/IP* protokolo pagalba, montuojant sistemą reikės atsižvelgti ir į tinklo tarp komponentų sudarymo galimybę (galima pasinaudoti jau egzistuojančiomis tinklo struktūromis sistemos montavimo vietoje arba sukurti dedikuotą tinklo struktūrą);
    d. Sistemos komponentų tarpusavio bendravimui naudojamas *Indy* komponentų rinkinys.
2. Portabilumas
    a. Kadangi sistema skirta sumontuoti ir naudoti vienoje vietoje, portabilumo reikalavimai jai nėra taikomi, prireikus sistemą perkelti į kitą vietą būtų atliekama standartinė demontavimo ir sumontavimo procedūra;
3. Paskirstymas
    a. Sistema iš esmės yra skirta naudoti viename pastate ar pastatų grupėje, tačiau pasitelkiant išorines priemones nekeičiant sistemos architektūros galima apimti ir didesnę teritoriją.

4. Pakartotinis panaudojimas
    a. Sistema yra kuriama „nuo nulio“, todėl joje pakartotino panaudojimo iš senesnių projektų nebus.
5. Projektavimo ir įgyvendinimo strategija
    a. Sistema kuriama ir projektuojama kaip vientisa, pateikiant užsakovui tik galutinį produktą, bet esant užsakovo pageidavimui teikiama informacija apie projekto kūrimo būseną;
    b. Sukūrus sistema per tam tikrą terminą ji sumontuojama pas pirkėją ir atliekamas galutinis sistemos patikrinimas, tik po to ji yra atiduodama eksploatuoti.
6. Projektavimo įrankiai
    a. UML diagramų sudarymo įrankiai, tokie kaip MagicDraw ar Microsoft Office Visio;
    b. C++ kalbos programų kūrimo ir derinimo aplinka Borland C++ Builder 6.0;
    c. Delphi kalbos programų kūrimo ir derinimo aplinka Borland Delphi 7.0;
    d. C# kalbos programų kūrimo ir derinimo aplinka Microsoft Visual Studio 2008;
    e. Vaizdo medžiagos peržiūros ir redagavimo programa IrfanView 4;
    f. Tekstinės medžiagos peržiūros ir redagavimo programa Notepad++
    g. Dokumentacijos ruošimo programa Microsoft Office Word;
    h. Operacinės sistemos Microsoft Windows įrankiai (Telnet, Process Manager, Explorer, Remote Desktop ir pan.).

4.2. Sistemos panaudojimo atvejų vaizdas



11 pav. ***Sistemos panaudos atvejų (use case) diagrama.***

4.2.1. Standartiniai sistemos panaudojimo scenarijai

1. Scenarijus 1:
    a. Vartotojas prieina prie kameros
    b. Kamera užfiksuoja veido atvaizdą, jį iškerpa, apdoroja ir perduoda centriniam valdymo moduliui atpažinti
    c. Centrinis valdymo modulis gavęs atvaizdą pasitelkdamas veido atpažinimo algoritmą jį (atvaizdą) palygina su duomenų bazėje esančių vartotojų atvaizdais;

   d. Gavęs labiausiai atitinkančio vartotojo ID sistemoje ir atitikimo kriterijų, centrinis valdymo modulis užklausia šio vartotojo teises iš duomenų bazės;
   e. Jei vartotojas yra atpažintas teisingai (atitikimo kriterijus viršija administratoriaus nurodytą slenkstį) ir turi teises patekti į patalpas, su kuriom susieta kamera, centrinis modulis perduoda komandą spynos moduliui atidaryti duris;
   f. Jei vartotojas buvo atpažintas teisingai, tačiau neturi teisių patekti į patalpas, centrinis modulis paneša vartotojui, kad jam į kambarį patekti negalima;
   g. Kitu atveju centrinis modulis nieko nedaro.
2. Scenarijus 2:
   a. Administratorius pasijungia prie sistemos;
   b. Pasirenka naujo vartotojo sukūrimo funkciją valdymo skydelyje;
   c. Suveda vartotojo duomenis (vardas, pavardė, kiti būtini duomenys, teisės patekti i tam tikras patalpas);
   d. Padaro vartotojo atvaizdą, kuris bus saugomas duomenų bazėje;
   e. Išsaugo visus duomenis sistemos duomenų bazėje;
   f. Atėjus numatytam laikui sistema atlieka vartotojų atvaizdų duomenų bazės regeneravimą (arba administratorius gali nurodyti, kad toks regeneravimas būtų atliktas nedelsiant)
   g. Naujas vartotojas gali naudotis sistema.

4.3. Didžiausi sistemos architektūros ypatumai

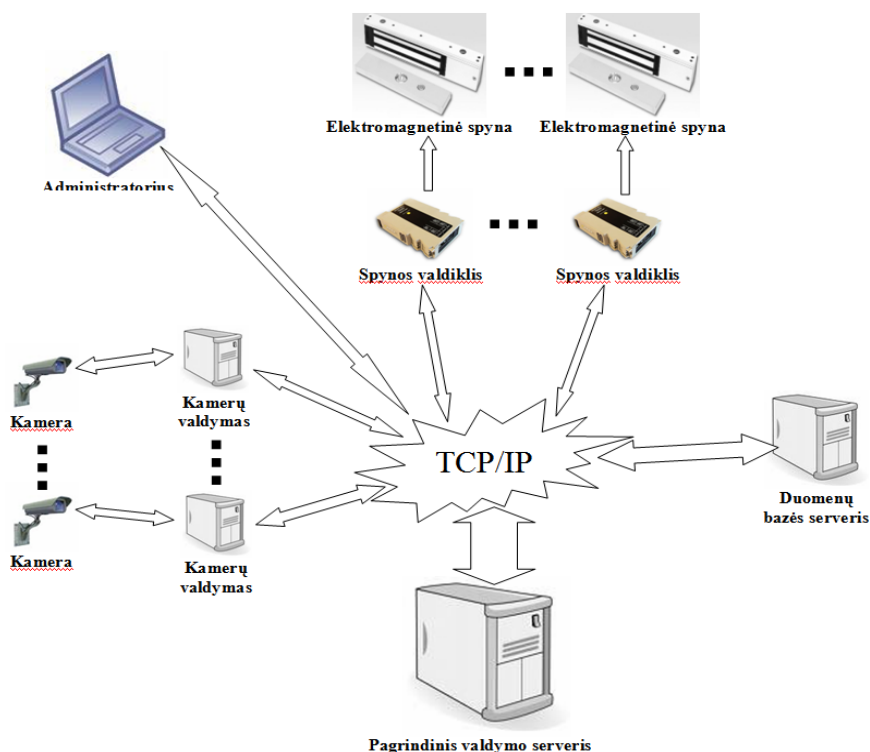
Projektuojant Veidą Atpažįstančią Įeigos Kontrolės Sistemą (toliau – VAĮKS), išnagrinėjus sistemos reikalavimus ir aptarus galimus sistemos architektūros modelius buvo nuspręsta pasitelkti vienu iš paprasčiausių ir labiausiai paplitusių paskirstytos sistemos architektūros modelį: *kliento-serverio*.

Kaip jau buvo minėta prieš tai buvusiuose skyriuose, kliento-serverio architektūra susideda iš vieno ar kelių serverių, kurie teikia tam tikras paslaugas prie jų prisijungusiems klientams, kurių kiekis dažniausiai nėra griežtai ribojamas. Serveriams gali talkinti specialūs papildomi procesai (middleware), kurių paskirtis yra paskirstyti apkrovą tarp serverių ar tiesiog apjungti visas serverių teikiamas paslaugas, neretai tokiu būdu virtualizuojant serverius tokiu būdu, kad klientams atrodo, jog paslaugas jiem tiekia tik vienas serveris. [17][18]

Taipogi middleware kartais teikia leidžia pasijungti prie specializuotų tinklinių servisų ar pateikia bendrą sąsaja keliems skirtingiems, bet panašios paskirties, servisams (pavyzdžiui, pateikia bendrą sąsaja bendravimui su keliom skirtingom duomenų bazių valdymo sistemom).[18]

Pasirinkome sistemos modulinę architektūrą, kai sistema susideda iš kelių atskirų modulių, kurie tarpusavyje sąveikauja per TCP/IP protokolą. Tai ne tik sumažina specializuotų komunikacijos sistemų panaudojimą diegiant sistemą (nes vietoj jų gali būti naudojama bet kokia infrastruktūra, palaikanti duomenų perdavimą TCP/IP protokolu, tokia kaip WI-FI, Ethernet ar net Internet), bet kartu ir įgalina komponentams bendrauti per didesnius atstumus, taip padidinant plotą, kurį gali aprėpti ši sistema. Taipogi modulinė sistemos architektūra įgalina modernizuoti ar net keisti individualius sistemos komponentus nemodifikuojant visos sistemos, ir net neišjungiant jos (nebent modernizuojami kertiniai komponentai ar duomenų bazė). Dar vienas didelis privalumas – galimybė plėsti sistemą neatliekant jokių pakeitimų programiniame kode, tiesiog prijungiant papildomus komponentus prie bendros sistemos.

Modulinė taipogi sąlygoja padidintą sistemos patikimumą, nes kiekvienas komponentas atsakingas tik už siaurą konkrečią veikimo sritį, todėl įvykus programinei nekritinėje sistemos dalyje maksimalus efektas, kurio galima tikėtis tokiu atveju – neveikiančios durys kurioje nors pastato dalyje. Kas būtų pataisoma tiesiog paleidus tą modulį iš naujo.



12. pav. ***Sistemos architektūros grafinis vaizdas.***

Kadangi sistema nėra mobili, tai pernešamumo kriterijai jau nėra taikomi. Tačiau, modulinės architektūros sistemą patogiau perkelti į naują vietą (pavyzdžiui pirkėjo įmonei kraustantis į naujas patalpas) nei vientisą.

Projektavimo metu buvo nuspręsta, kad

sistemą bus sudaryta iš keturių pagrindinių tarpusavyje susietų komponentų:

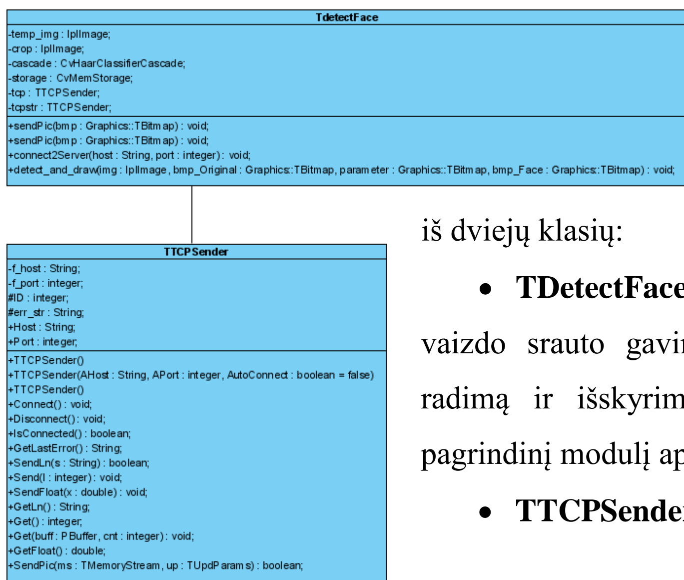
- **Kameros valdymo modulis.** Šis modulis apdoroja iš kameros (ar kelių kamerų) gautą vaizdą, suranda vaizdo sraute žmogaus veido atvaizdą, jį iškerpa ir persiunčia pagrindiniam valdymo moduliui apdoroti. Šiame modulyje vaizdo srauto apdorojimui ir veido atvaizdo atpažinimui pritaikyta *Intel OpenCV* biblioteka.
- **Pagrindinis valdymo modulis** koordinuoja visų komponentų darbą, bei tarnauja kaip visos sistemos serveris, t. y. kiti sistemos komponentai jungiasi prie centrinio modulio, kad perduoti jam duomenis ir/ar gauti iš jo komandas. Šis modulis be koordinavimo atlieka ir kitą labai svarbų darbą – gavęs iš kameros valdymo modulio iškirptą veido atvaizdą, jis jį apdoroja ir pasitelkdamas veido atpažinimo algoritmą pagal duomenų bazėje išsaugotus vartotojų atvaizdus. Pagal veido atpažinimo rezultatus pagrindinis valdymo modulis nurodo spynos, susietos su kamera, iš kurios buvo gautas vaizdas, valdikliui atidaryti arba neatidaryti durų.
- **Spynos valdymo modulis** priima komandas iš pagrindinio valdymo modulio bei gali turėti alternatyvias galimybes autentifikuoti vartotoją, jei šis nebūtų atpažintas automatiškai (ar papildomai prie atpažinimo, jei reikia ypatingo saugumo).
- **Administratoriaus valdymo skydelio modulis** pateikia sistemos administratoriui rinkinį įrankių, kurių pagalba šis gali konfigūruoti, valdyti ir diagnozuoti sistemą. Taipogi šis modulis, nors ir nedalyvauja tiesiogiai sistemos veikime, gali įtakoti sistemos darbą: jo pagalba galima įtraukti naujus kameros ir spynos modulius, naujus sistemos vartotojus, sustabdyti ir paleisti sistemą bei tiesiogiai bendrauti su duomenų baze, vykdyti joje užklausas.
- Visi sistemos komponentai yra nepriklausomi, ir, jei nesikeičia sąsaja tarp komponentų, vieno iš jų pakeitimas ar modifikavimas neįtakoja kitų komponentų darbo.

4.4. Komponentų detalizavimas

4.4.1. Kameros modulis

Kameros valdymo modulis sudarytas iš dviejų klasių:

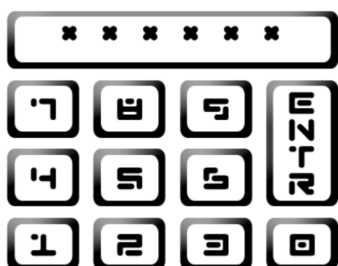
- **TDetectFace** klasė atsakinga už darbą su kamera, kameros vaizdo srauto gavimą, apdorojimą, veido atvaizdo (ar atvaizdų) radimą ir išskyrimą. Išskyrus veido atvaizdą jis siunčiamas į pagrindinį modulį apdorojimui.
- **TTCPSender** klasė atsakinga už ryšio palaikymą su



13 pav. ***Kameros modulio klasių diagrama.***

pagrindinių moduliu bei išskirtų atvaizdų perdavimu TCP protokolu pagrindiniam moduliui apdoroti.

4.4.2. Spynos modulis



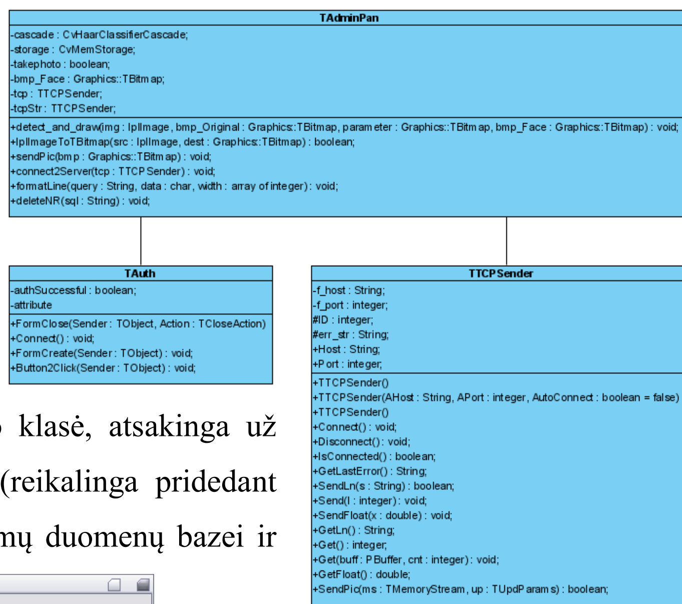
14 pav. *Spynos modulio grafinės sąsajos atvaizdas.*

Spynos modulis, kaip ir kameros, yra sudarytas iš dviejų klasių:

- **TLock** – grafinė sąsaja su vartotoju (atvaizduoja skaičių klaviatūrą PIN kodo įvedimui bei pateikia vaizdinę informaciją, leidžiama vartotojui įeiti į patalpas, ar ne.
- **TTCPSender** – klasė, bendra trims iš keturių sistemos modulių, kurioje aprašytas tinklo protokolas ir pateikiami metodai ryšio užmezgimui tarp modulių TCP/IP protokolu ir duomenų mainams. Ši klasė atsakinga už sąsaja tarp elementų.

4.4.3. Administratoriaus valdymo skydelis

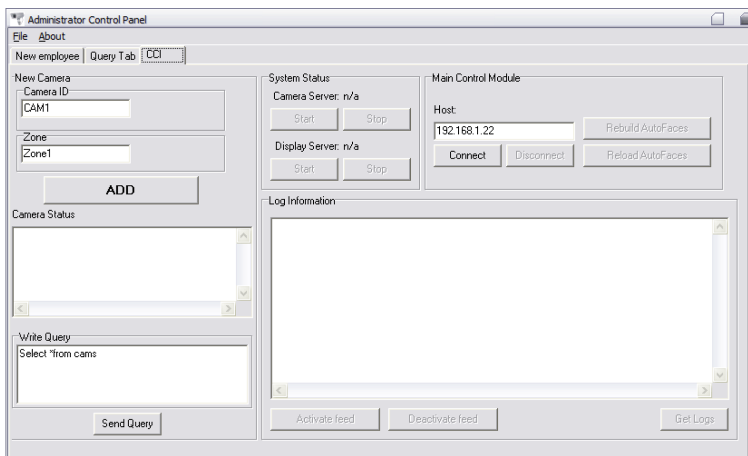
Administratoriaus skydelis pateikia sistemos valdymo, naujų vartotojų kūrimo, redagavimo, šalinimo įrankius, modulių pridėjimo/šalinimo galimybę bei tiesioginę sąsaja su duomenų baze, leidžiančia vykdyti užklausas tiesiogiai duomenų bazėje.



15 pav. *Administratoriaus valdymo skydelio klasių diagrama.*

Modulį sudaro trys klasės:

- **TAdminPanel** – pagrindinė skydelio klasė, atsakinga už grafinę vartotojo sąsaja, kameros valdymą (reikalinga pridedant naują vartotoją į sistemą), duomenų, siunčiamų duomenų bazei ir gaunamų iš jos apdorojimą ir pateikimą administratoriui, vartotojų komandų transformavimą į sistemai suprantamas komandas. Taipogi ši klasė surenka informaciją iš kitų komponentų apie sistemos būklę, atvaizduoja pagrindinio



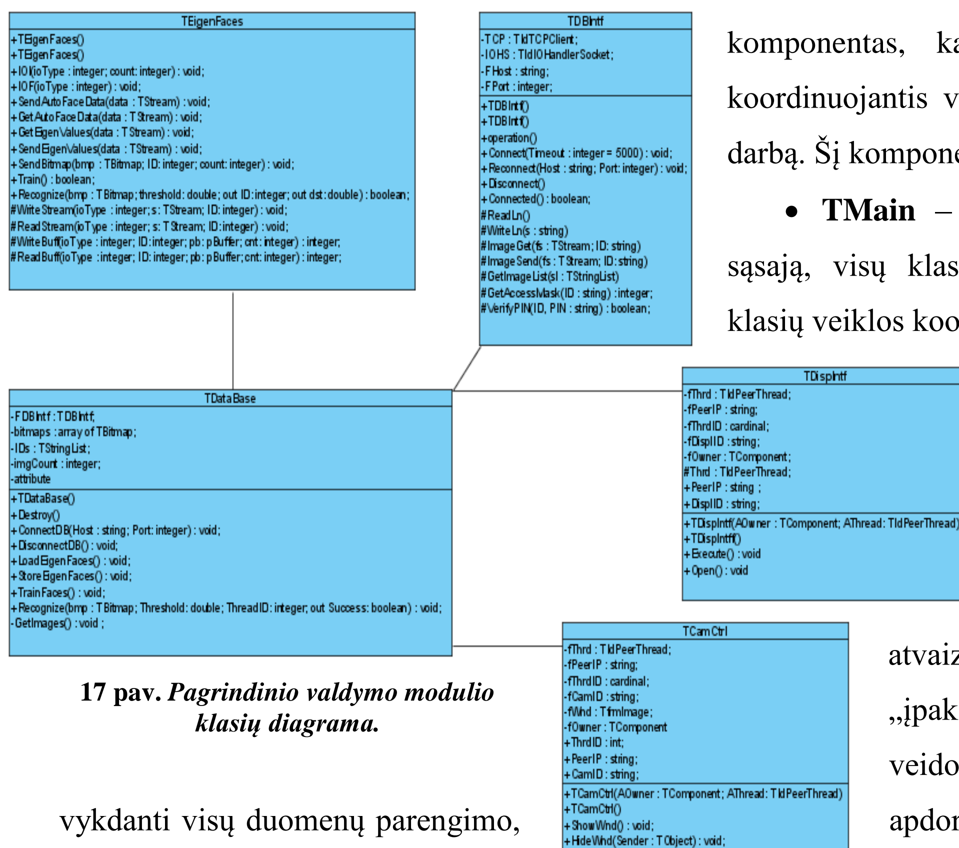
16 pav. *Administratoriaus skydelio grafinės sąsajos vaizdas.*

valdymo modulio žurnalo (*log*) informaciją ir leidžia sustabdyti ar paleisti sistemą ar inicijuoti rankinį atvaizdų regeneravimo procesą.

- **TAuth** klasė atsakinga už administratoriaus autentifikaciją sistemoje. Ji pateikia vartotojo vardo ir slaptažodžio užklausos dialogą bei perduoda šiuos duomenis duomenų bazės moduliui, kad tas verifikuotų vartotoją.
- **TTCPSender** klasė pateikia tinklo protokolo metodus valdymo skydelio kitoms klasėms. Ji kuria pasijungimus prie kitų modulių, perduoda ir priima informaciją tarp modulių.

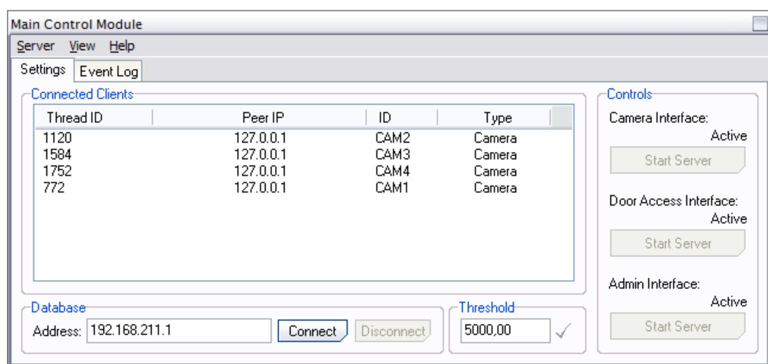
4.4.4. Pagrindinis valdymo modulis

Tai – pagrindinis sistemos komponentas, kaip dirigentas valdantis ir koordinuojantis visos sistemos nepriekaištingą darbą. Šį komponentą sudaro daugelis klasių:



17 pav. *Pagrindinio valdymo modulio klasių diagrama.*

- **TMain** – klasė, atsakingą už grafinę sąsają, visų klasių sukūrimą, gijų valdymą, klasių veiklos koordinavimą.
- **TEigenFaces** – ši klasė vykdo gauto atvaizdo atpažinima pasitelkiant duomenų bazėje jau esančius atvaizdus. Ši klasė tarnauja kaip „įpakavimas" (*angl. wrapper*) veido atpažinimo algoritmui, apdorojimo ir rezultatų patikrinimo vykdanti visų duomenų parengimo, funkcijas.
- **TDBIntf** – klasė, skirta darbui su duomenų baze. Pateikia metodus darbui su viena duomenų baze: prisijungimas, užklausų vykdymas, duomenų gavimas ir siuntimas ir t.t. Tai duomenų bazės sąsajos klasė.
- **TDispIntf** – klasė, skirta darbui su spynos moduliais. Tarnauja kaip sąsajos objektas tarp pagrindinės



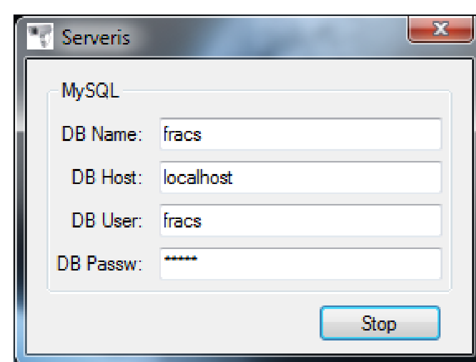
18 pav. *Pagrindinio valdymo modulio vartotojo sąsajos atvaizdas.*

darbo klasės ir spynos modulio. Turi savyje metodus tinklo sąsajai su spynos valdikliu.

- **TCamCtrl** klasė aprašo sąsajos su kameros moduliu metodus. Pateikia duomenų gavimo, identifikavimo metodus.
- **TDataBase** – viena iš pagrindinių komponento klasių. Ši klasė „orkestruoja“ pagrindinį sistemos darbą – duomenų gavimą ir perdavimą tarp kameros, spynos ir duomenų bazės modulių, taipogi duomenų, gautų iš kameros, apdorojimą ir pateikimą atpažinimo algoritmui, atpažinto vartotojo teisių nustatymą ir su kamera susieto spynos modulio kontrolė atpažinimo atveju.

4.4.5. Duomenų bazės modulis

Duomenų bazės modulis buvo kuriamas *Microsoft Visual Studio 2008* pagalba ir rašomas *C#* kalba. Jį sudaro tik viena klasė, atsakinga už bendravimą su kitais sistemos komponentais ir tarpininkaujanti tarp sistemos ir duomenų bazės. Darbui su duomenų baze naudojamas *MySQLConnector* komponentas, kuris leidžia atlikti visus reikalingus veiksmus su duomenimis – įtraukti, redaguoti, skaityti ir šalinti.



19 pav. ***Duomenų bazės grafinės sąsajos vaizdas.***

4.5. Automatinis kodo generavimas

Kuriant sistemą buvo plačiai taikomi automatinio kodo generavimo įrankiai.

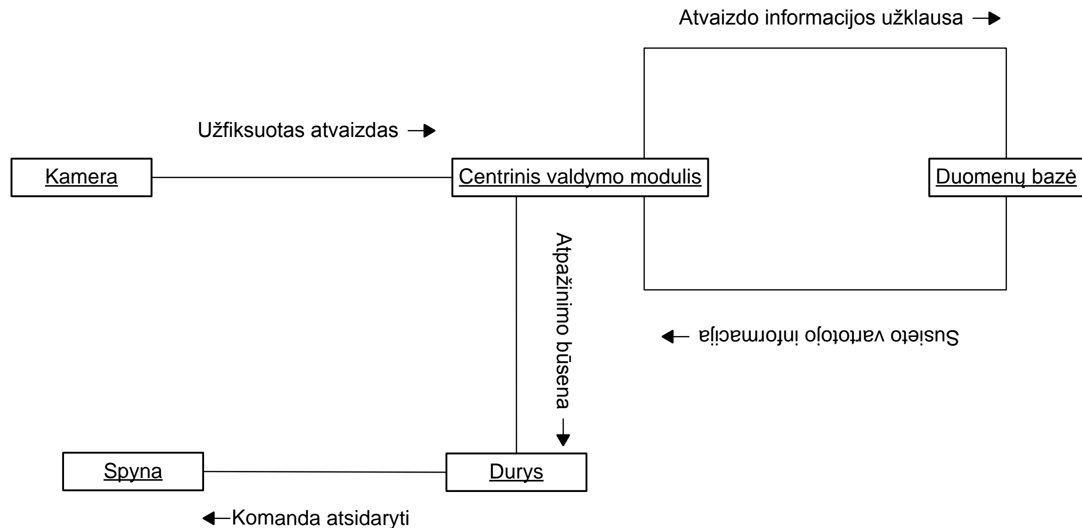
Pagrindinis iš tokių įrankių, kuris yra visuose naudotose programų kūrimo aplinkose, yra automatizuotas vizualus vartotojo grafinės sąsajos redaktorius. Jis leidžia pelės pagalba išdėstyti komponentus programos grafinėje sąsajoje taip, kaip jie turi atrodyti. Programa pati sugeneruoja atitinkama kodą šiems komponentams sukurti, išdėstyti ir jų funkcionalumui užtikrinti. Programuotojui tereikia susieti įvykių (*events*) metodus su komponentų įvykiais (pavyzdžiui, *OnClick* įvykis iškviečiamas tada, kai vartotojas pele spragteli komponentą, pvz., mygtuką). Tačiau net ir tai yra automatizuota, programuotojui tereikia pasirinkti iš sąrašo norimą įvykį ir dvigubai spragtelėti jį pele, aplinka pati sugeneruos tinkamą įvykio metodą ir susies jį su komponentu.

Be automatizuoto grafinės sąsajos kūrimo dar buvo taikytos priemonės metodų aprašymų generavimui, kalbos konstrukcijų (*klasių, struktūrų, tipų aprašai, if-else, switch, case sakiniai ir pan.*), automatinio kodo pildymo įrankiai, kuriuos pateikia pati programų kūrimo aplinka. Iš UML klasių diagramų buvo sukurti klasių aprašymai, kuriuos tereikėjo papildyti veikimo logika, pagal duomenų bazės struktūrinę diagramą buvo automatiškai sugeneruotas duomenų bazės modelio kodas.

Taipogi buvo taikomi ir automatizuoto kodo pertvarkymo (*refactoring*) įrankiai.

4.6. Sistemos dinaminis vaizdas

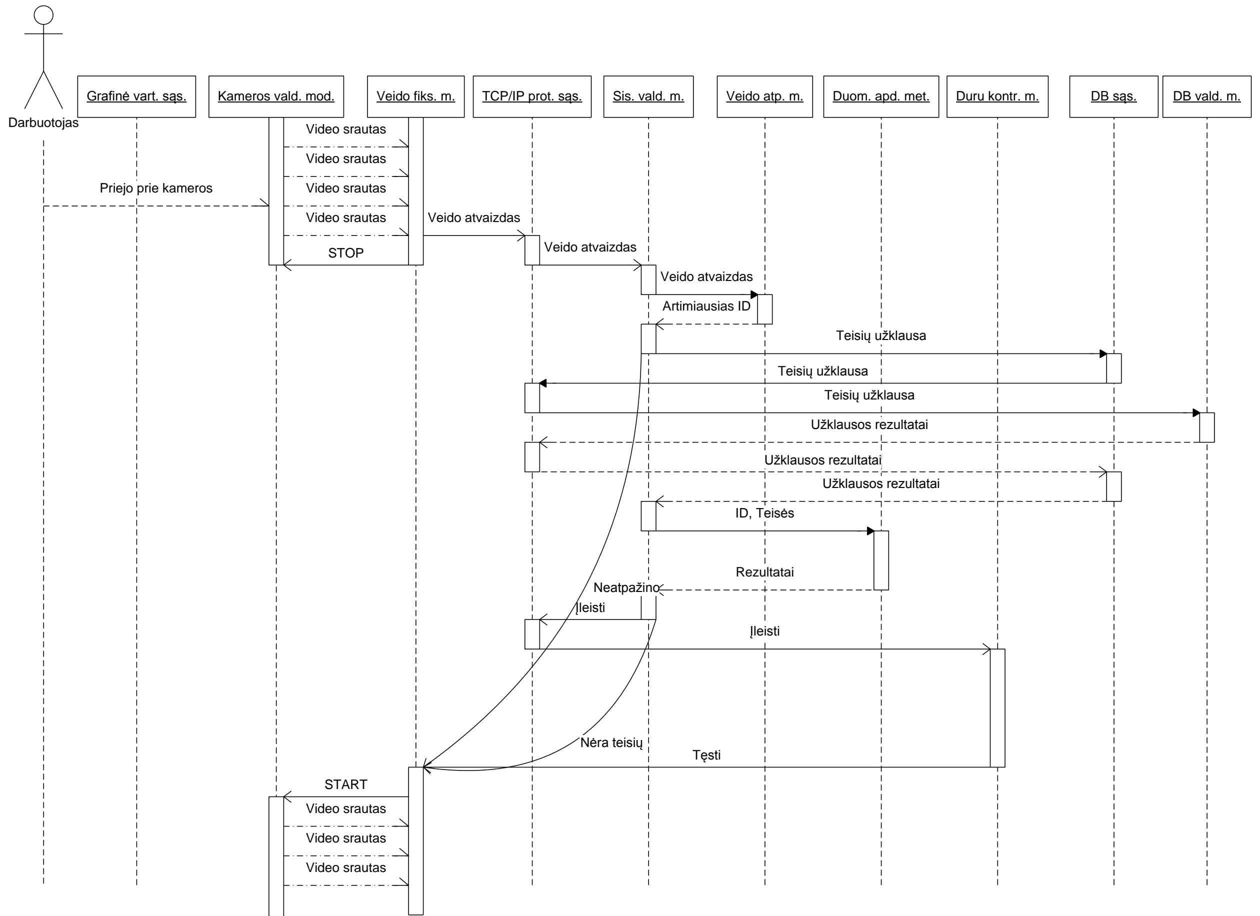
4.6.1. Bendradarbiavimo diagrama



20 pav. ***Sistemos bendradarbiavimo diagram.***

Kamerai užfiksavus tinkamą atpažinimui atvaizdą jis yra iškerpamas iš bendro kadro, apdorojamas ir perduodamas centriniam moduliui atpažinti. Centrinis modulis apdorojęs atvaizdą ir palygines su kitais turimais nustato panašiausio atvaizdo savininką ir užklausia jo informacijos iš duomenų bazės. Gavęs informaciją nustato, ar yra suteikta atpažintam vartotojui patekti į patalpas, kurių kamera perdavė užklausą, ar ne. Rezultatai perduodami spynos valdymo moduliui, kuris arba atvaizduoja informaciją, kad vartotojui uždrausta patekt į patalpas arba atidaro durys jei vartotojui leista patekti į patalpas.

4.6.2. Vartotojo atpažinimo ir įleidimo į patalpas sekų diagrama



21 pav. ***Vartotojo atpažinimo ir įleidimo į patalpas veiksmo sekų diagrama.***

Pagrindinė, ir svarbiausia, sistemos savybė – tai atpažinti vartotoją ir pagal jam suteiktas privilegijas jį arba įleisti į tam tikras įmonės patalpas arba ne. Be to tai vienintelė sistemos funkcija, kuri tiesiogiai naudojama visų įmonės darbuotojų. Vartotojas turi prieiti prie kameros kad ta jį galėtu nufotografuoti, tada iš paveiksliuko yra išskiriamas veido atvaizdas, persiunčiamas į centrinį valdymo modulį, kur jis yra apdorojamas, atpažįstamas, nustatomos šiam vartotojui nustatytos privilegijos ir pagal tai nusprendžiama ar įleisti vartotoją į patalpas ar ne.

Kameros valdymo modulis siunčia duomenų srautą be sustojimo ir *asinchronishkai*, dėl to įvykus veido *detektavimui* kadre srautas turi būti stabdomas specialiu pranešimu. Taipogi sekų diagramoje parodyti alternatyvūs keliai esant neatpažinimo ir teisių neatitikimo situacijoms (kreivės). Pasibaigus atpažinimo procesui kameros duomenų srautas turi būti paleistas per naujo.

5. Paskirstytos sistemos VAĮKS testavimo metodikų taikymas ir tyrimas

5.1. Testavimo metodikų aprašymas ir taikymas

Šiame skyrelyje aprašysiu kaip gali būti ir buvo pritaikytos skirtingos testavimo metodikos kuriant paskirstytą sistemą *Veidą Atpažįstanti Įeigos Kontrolės Sistema*. Vienos iš jų buvo labiau sėkmingai taikytos, kitos iš jų mažiau.

5.1.1. Bendra testavimo metodika

Sistemos testavimas buvo planuojamas ir vykdomas praktiškai nuo pat projekto pradžios. Vos suderinus sistemos architektūrą buvo pradėti kurti *kamščiai*, skirti testuoti pavienes sistemos dalis bei suderinti tinklo protokolų veikimą.

Kamščiai dar yra labai svarbus tokio kūrimo proceso dalis todėl, kad jie leidžia testuoti sistemos komponentus ne tik nesant užbaigtai visai sistemai, tačiau specifiškai parašyti kamščiai įgalina pratestuoti komponentų veikimą už jų dokumentuotų savybių ribų, nes kamščiui galima suprogramuoti kiek kitokią ar platesnę sritį apimančia logiką, nei komponento, kurį pavaduoja kamštis. Tačiau noriu atkreipti dėmesį, kad mūsų projekte kamščiai apjungia tiek kamščio, tiek *valdiklio* vaidmenį, nes jie vienu metu pavaduoja komponentą supaprastindami jo funkcijas ir pateikia funkcionalumą valdyti testuojamąjį komponentą. Todėl *kamščiu* toliau vadinsiu laikiną sistemos komponentą, pavaduojantį sistemos komponentą testavimo metu ir turintį ribotą to komponento funkcionalumo dalį.

Dar vienas kamščių panaudojimo būdas paskirstytų sistemų testavime – tinklo ar atskirų komponentų trikčių testavimas. Pavyzdžiui, kamštį galima suprogramuoti, kad nutrauktų duomenų perdavimą jų neužbaigęs, ir stebėti sistemos elgesį šiuo atveju. Arba, kamštis gali užklausti duomenų persiuntimą, tačiau nepriimti. Abejais šiais atvejais sistema rizikuoja tapti mažų mažiausia nestabilia, jei ji nebuvo suprogramuota susidoroti su šiom situacijom.

Vėliau kamščio kodas gali būti įtraukiamas į kuriamą komponentą, taip sutaupant ne tik laiką, bet ir sumažinant klaidų galimybę, nes kamščio kodas neretai būna gerai ištestuotas su kitais realiais komponentais.

Nors pradėję kurti sistemą mes dar nebuvome susipažinę su *kūrimo testuojamumui* koncepcija, tačiau dabar, peržiūrint atliktą darbą, galima pastebėti esminius šios koncepcijos principus sistemos dizaine: aiškios sąsajos tarp komponentų (ir komponentų viduje tarp klasių), sklandžiai išdėstyti ir lengvai suprantami reikalavimai funkcionalumui, aiškiai parašytas kodas. Tai

gali būti įtakota to, kad sistemos architektūra buvo sukurta taip, kad palengvinti kamščių naudojimą testuojant.

Be kamščių, sistemos testavimui buvo pasitelktos ir kitokios testavimo metodikos: vienetų ir klasių testai, statinė kodo analizė, kolegų peržiūrą (kai vienas kūrėjas peržiūri kito kodą ieškodamas galimų klaidų, „pavojingų" struktūrų ir neaiškių vietų).

Vienetų ir klasių testavimas taikomas lokaliai kiekviename komponente jau jo kūrimo eigoje. Sukūrus klasę ir apsirašius metodus (dar projektavimo/modeliavimo metu) yra aprašomi ir testai klasės metodams, kurie apima standartines reikšmes ir patikrina metodų grąžinamus duomenis pagal iš anksto numatytą duomenų rinkinį. Toks testavimas gali būti atliekamas ir po pakeitimų moduliuose, kad įsitikint, jog funkcionalumas nebuvo pažeistas (*regresinis testavimas*). Ši metodika buvo pritaikyta pagrinde veido atpažinimo algoritmui testuoti.

Statinė kodo analizė yra bene labiausiai automatizuota programų kūrimo ir testavimo dalis. Tačiau šis testavimo metodas gali atrasti ir padėti ištaisyti tik nedidelę grupę klaidų, susijusių su neteisingai naudojamomis kalbos konstrukcijomis, neinicializuotais kintamaisiais ar galimomis indeksavimo klaidomis. Dauguma šiuolaikinių aplinkų gali atlikti statinę kodo analizę fone, programuotojui dar tik rašant kodą ir pranešti apie potencialias klaidas. Be to, daugelis kompiliatorių taipogi atlieka statinę kodo analizę prieš kompiliuodami kodą. Mūsų naudoti programavimo įrankiai palaiko statinę kodo analizę ir padėjo mums išvengti daugelio potencialių problemų.

Kolegų peržiūra (*peer review*) yra neautomatizuojama testavimo metodologija, dėl to ji atima nemažai laiko, tačiau neretai gali atskleisti tokias klaidas, kurių nerastų kitos testavimo metodikos, dar jom net nepasireiškus. Šis metodas remiasi kodo peržiūra, kurią atlieka kitas žmogus, nei kad parašęs šį kodą. Neretai prieš atliekant tokias peržiūras yra sudaromas sąrašas galimų klaidų ir pavojingų kodo konstrukcijų, kurių reiktų vengti. Taipogi peržiūrėjęs kodą kolega gali pasiūlyti alternatyvius, efektyvesnius ar saugesnius problemų sprendimo būdus. Ši metodologija buvo taikoma ir mūsų projekte, kai kolegos peržiūrėdavo vienas kito kodą ir kartais pasiūlydavo alternatyvių sprendimo būdų ar tiesiog pagelbėdavo klaidų paieškoje.

5.1.2. Kameros modulio testavimo metodika

Kameros modulio kūrimas ir testavimas susidėjo iš kelių fazių. Pirmojoje fazėje buvo vykdomas vaizdo gavimo iš kameros ir jo atvaizdavimo lange testavimas: ar modulis tvarkingai prisijungia prie kameros, ar gaunamas vaizdo srautas atitinka kameros siunčiamą. Prisijungimui prie

kameros naudojama *OpenCV* biblioteka. Testavimui buvo naudojama Logitech USB web kamera. Kameros vaizdo srautas atvaizduojamas atskirame lange. Šiame etape testavimas apėmė statinę analizę ir programos derinimą pasitelkiant programavimo aplinkos įrankius.

Toliau buvo tikrinamas veido išskyrimo algoritmo veikimas ir derinimas – ar teisingai yra atpažįstamas veidas, ar tvarkingai programa jį randa sraute, kaip programa susidoroja, jei vaizdo sraute figūruoja daugiau nei vienas veidas, renkama informacija, kiek vidutiniškai užtrunka veido radimas sraute ir pan. Parametrai veido radimui (*detection*) vaizdo sraute saugomi *XML* formatu faile. Testavimas įtraukė derinimą su derinimo įrankiais, *XML* failo duomenų modifikavimą teksto redaktoriumi ir testavimą paleidžiant programą su USB web kamera ir išskyrimo rezultatų stebėjimą derinimo įrankiais.

Paskutinėje fazėje reikejo prijungti tinklo kodą ir ištestuoti jau užbaigto komponento veikimą įvairiomis sąlygomis. Kadangi sistemos kūrimo metu buvo plačiai taikomas *kamščių ir valdiklių* testavimo metodas, tai šioje stadijoje buvo panaudotas tinklo sąsajos kodas iš pagrindiniam moduliui skirto *kamščio*, pavaduojančio kameros modulį. Tereikėjo suderinti grafikos formatus, nes *OpenCV* naudoja nuosavą vidinį paveiksliukų formatą *cvBitmap*, o pagrindinis valdymo modulis reikalauja atvaizdų *Windows Bitmap* formatu.

22 pav. ***Kameros modulio testavimas. Ekrano vaizdas.***

Tinklo testavimui buvo naudojamas kamštis, pavaduojantis pagrindinį valdymo modulį. Šis kamštis priima pasijungimus iš kamerų, atvaizduoja jų persiustus atvaizdus ir parodo papildomą informaciją (kiek užtruko persiuntimas, atvaizdo dydis ir pan.). Kamščio kodas vėliau įtrauktas į pagrindinio modulio kodą, tad kameros modulio integravimas į sistemą įvyko sklandžiai.

5.1.3. Spynos modulio testavimas

Kadangi spynos modulis buvo kuriamas paskutinis, tai tuo metu jau buvo sukurti pilnai funkcionuojantys kameros bei pagrindinis moduliai, todėl šiam moduliui nebuvo taikomas *valdiklių ir kamščių* testavimo metodas. Tinklo protokolui palaikyti buvo panaudota klasė iš kitų komponentų, kuri jau buvo ištestuota ir irgi ypatingo testavimo nereikalavo.

Testuojama buvo grafinė sąsaja – ar visi mygtukai tvarkingai veikia, ar grafiniame lauke teisingai atvaizduojama visa informacija, ar programa teisingai interpretuoja serverio siunčiamus duomenis. Šis testavimas buvo atliktas pasitelkiant programavimo aplinkos derinimo įrankius, paleidžiant programą ir tikrinant (kartais net pažingsniui vykdant programą su derinimo įrankiu). Kadangi šis sistemos komponentas yra pats paprasčiausias struktūriškai, jam ypatingų testavimo metodikų nebuvo panaudota.

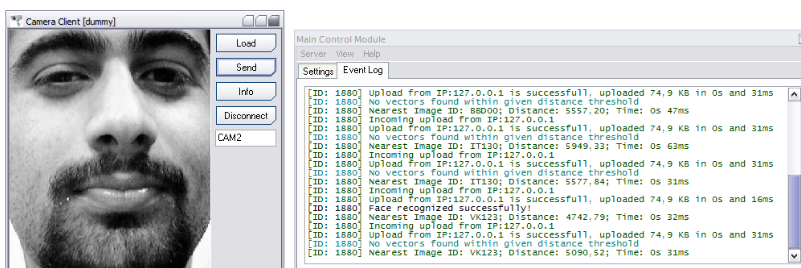
5.1.4. Administratoriaus skydelio testavimas

Administratoriaus valdymo skydelis yra savotiškai unikalus sistemos komponentas. Tai vienintelis komponentas, kuris yra neprivalomas normaliam sistemos darbui, be to, tai vienintelis komponentas, kurio visos funkcijos reikalauja vartotojo veiksmų. Būtent dėl to didžiausia šio komponento funkcionalumo testavimo ir derinimo dalis buvo atliekama neautomatizuotu būdu.

Šis komponentas yra sukurtas duomenų bazės *kamščio* pagrindu, išplečiant jį papildomu funkcionalumu, praplečiant galimybes. Kameros valdymo kodas, kuris naudojamas naujo vartotojo registravimo metu gauti jo atvaizdą tam, kad įrašyti į duomenų bazę, paimtas iš anksčiau sukurto kameros modulio, kartu su tinklo komponentu *TTCPSender*. Daugiausia dėmesio susilaukė sąsajos su duomenų baze testavimas (reikėjo įsitikinti, kad duomenys yra teisingai perduodami ir atvaizduojami) bei sąsaja su pagrindiniu valdymo moduliu. Naujo vartotojo sukūrimo ir esamų vartotojų redagavimo funkcionalumas paremtas duomenų bazės sąsaja. Šios sąsajos testavimas buvo atliekamas pusiau automatiniu būdu panašiai į vienetų (*unit*) testą – sukuriamas pagal duomenų bazės turinį automatizuotas testas, kuris po to vykdomas, kad patikrinti komponento veikimo teisingumą.

Šis komponentas atsakingas už didžiąją dalį klaidų, pasitaikiusių sistemoje kūrimo metu.

5.1.5. Pagrindinio valdymo modulio testavimas



23 pav. ***Pagrindinio valdymo modulio testavimas kamščio pagalba.***

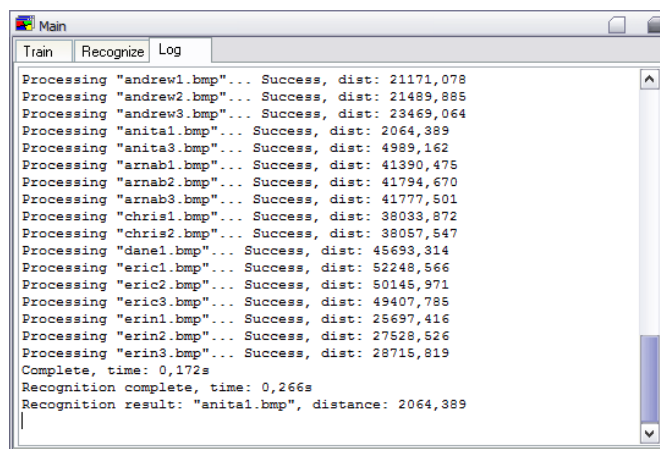
Pagrindiniam moduliui testuoti buvo sukurtas kameros modulio *kamštis* ir laikinai išjungtas (pakeistas vidiniu *kamščiu*) duomenų bazės funkcionalumas. Tuo metu veido atpažinimas vykdomas iš išankstо paruoštų duomenų rinkinio. Kadangi kameros *kamštis* idealiai replikuoja kameros modulio funkcionalumą, vėliau integruojant su realiais komponentais nesklandumų nekyla. Su duomenų

baze susijęs funkcionalumas testuojamas su „gyva“ duomenų baze, pasitelkiant kūrimo aplinkos derinimo įrankius (*debugger*).

Testavimo atžvilgiu šis komponentas yra sudėtingiausias, nes jį sudaro trys serveriniai moduliai (*kamerų, spynų* ir *administratoriaus skydo* pasijungimui). Kiekvienas iš šių modulių sukuria kelias programines gijas, dėl to jis gali vienu metu lygiagrečiai aptarnauti gan didelį skaičių klientų. Didžiąją dalį laiko šios gijos praleidžia laukdamos duomenų iš klientų (atvaizdo iš kameros, PIN kodo iš spynos modulio, komandos iš administratoriaus skydelio).

Didžiąją dalį testavimo šiam moduliui teko atlikti pasitelkiant aplinkos derinimo įrankį, nes reikėjo atsižvelgti ir į tarp gijų vykstančius duomenų mainus bei jų sąveiką. Kadangi gijos dalinasi bendra atmintimi reikėjo atidžiai ištestuoti, ar tarp jų veikimo procese nekyla konfliktų. Kadangi pagrindiniai duomenys, kuriuos gijos kečia, yra susijęs su atpažinimo algoritmu, buvo padarytas mechanizmas, kad tik viena gija vienu metu gali vykdyti atpažinimo operaciją (atpažinimo klasė realizuota kaip seniūnas). Kadangi pačio algoritmo veikimas buvo testuojamas atskirai, šios klasės testavimas apsiribojo seniūno objekto testavimu kai į jį kreipiasi daugiau nei viena gija.

Algoritmui testuoti buvo sukurta atskira programa, kuri atlikdavo algoritmo *vienetų testavimą*, kartu taikydama *profiliavimo* metodus, kad surinkti informaciją apie algoritmo veikimo greitį, siekiant paspartinti programos veikimą. Programa pati gali sukurti testini rinkinį iš vartotojo parinktų duomenų ir atlikti algoritmo testavimą bei atrasti klaidas, kuruos gali atsirasti modifikuojant algoritmą.



24 pav. ***Vienetų testavimo programos grafinė sąsaja.***

Testavimui paruošti 6 skirtingi testiniai rinkiniai, kurių suminis kodo padengimas yra 100%. Be to buvo atliekamas ir klasės mutacinis testavimas, kas nežymiai pakeičiama kuri nors algoritmo dalis (pavyzdžiui vietoj kintamųjų sudėties taikoma atimtis) ir stebimi testavimo rezultatai. Visų pakeitimų atvejų pokyčiai buvo rasti, tačiau kartu klaidas grąžindavo ir kiti metodai, kurie naudoja klaidą turinčių metodų rezultatus.

5.1.6. Duomenų bazės modulio testavimas

Didžioji dauguma šio modulio testų taip pat buvo atlikta pasitelkiant programų kūrimo aplinkos įrankius. Buvo pritaikyti statinės analizės metodai iš duomenų bazės gaunamų duomenų

apdorojimo algoritmam, nemažai rankinio derinimo su derinimo įrankiu ir *SQL* užklausų profiliavimo įrankis užklausom optimizuoti ir struktūrizuoti.

Su duomenų baze buvo susijusi viena didžiausių ir mįslingiausių sistemos klaidų – kartais po naujo vartotojo įtraukimo į sistemą sistema negalėdavo regeneruoti atvaizdų galerijos kol nebūdavo perkrauti visi sistemos moduliai. Paleidimas pasitelkiant derinimo įrankius tiek centriniame modulyje, tiek duomenų bazės modulyje nieko neparodydavo – sistema tokiu atveju veikdavo tvarkingai. Negana to, pati klaida retai pasirodydavo toje pačioje vietoje, bet visada jos požymis būdavo klaidos pranešimas „*connection closed gracefully*“ pagrindinio modulio lange. Po kelių dienų įnirtingo medžiojimo pavyko atrasti, kad klaida buvo sąlygojama trijų veiksnių: *CLR Garbage Collector* (.NET karkaso šiukšlių surinkėjas, atsakingas už nebenaudojamų atminties struktūrų išvalymą) veikimo, failų sistemos kreipinių ypatumų ir nedidelės klaidos duomenų persiuntimo kode iš duomenų bazės pusės. Šios klaidos priežastys buvo nustatytos tik po to, kai visi komandos nariai atidžiai peržiūrėjo programos kodą ir pagal sudarytą tikrinimo sąrašą atliko kodo analizę. Po to pašalinti šią klaidą nebuvo sunku.

Garbage Collector sąlygojo, kad failo rašymo objektas nebūdavo laiku išvalomas, dėl ko programa negalėdavo atsidaryti failo skaitymui, kildavo programos išimties klaida (*exception*) ir gija, kuri aptarnaudavo pagrindinio modulio prisijungimą TCP/IP protokolu būdavo nutraukiama be perspėjimo, dėl ko pagrindinis modulis sekantį sykį bandydamas susijungti su duomenų bazę per nutrauktą ryšį (nes nežinojo, kad jis buvo nutrauktas) gaudavo klaidos pranešimą. Klausimas, kodėl viskas veikdavo tvarkingai, kai programa būdavo pristabdoma (*break*) ir peržiūrima pažingsniui (*stepping*), tikriausiai susijęs su *Garbage Collector* veikimu, kuris, matyt, veikia skirtingai normalaus paleidimo metu ir programos pristabdymo metu. Ši klaida, galų gale, buvo ištaisyta programiškai priverčiant *Garbage Collector* išvalyti visus su failais susijusius objektus iškart po jų panaudojimo.

5.1.7. Integracinis testavimas

Po to, kai visi komponentai užbaigti, juos reikia sujungti tarpusavyje. Tačiau, kad ir kaip detaliai būtų ištestuoti komponentai, visada atsiranda nesklandumų juos apjungiant. Apjungiant komponentus taikomi kamščių, juodos ir baltos dėžių testavimo metodai. Kamščiai pavaduoja dar nepajungtus komponentus, o pajungti testuojami pagal jų funkcionalumą (juodos dėžės) ir pagal jų vidinę logiką (baltos dėžės). Baltos dėžės testavimo metu gali būti pasitelkiami ir programavimo aplinkos derinimo įrankiai.

Kadangi realiuose komponentuose buvo taikomas kodas iš kamščių, integruojant sistemą daugelio nesklandumų, susijusių su tinklu ir bendravimo protokolais pavyko išvengti. Didžiausia dalis iškilusių nesklandumų buvo susiję su vidine komponentų logika, kuri kartais įtakojo ir protokolo veikimą. Kaip pavyzdį pateiksiu situaciją, kai neteisingai konvertuotas atvaizdas iš *OpenCV* formato į *Windows Bitmap* formatą destabilizavo sistemos darbą.

Sistemoje perduodami atvaizdai turi būti griežtai apibrėžto formato, kitaip veido atpažinimo algoritmas gali grąžinti nenuspėjamus rezultatus. Atvaizdai privalo būti 150x150 pikselių dydžio, nespalvotas (*grayscale*), tačiau išsaugotas 8-bit RGB formatu, kai vienam pikseliui (256 pilkos spalvos atspalviu) skiriama 3 baitai (t.y. viena reikšmė dubliuojasi per tris baitus). Tokio formato *Windows Bitmap* paveiksliukas užima 87,9 KB dydžio. Atvaizdas tinkle perduodamas 5 KB dydžio paketais. Kadangi duomenų bazės modulis parašytas *C#* kalba, paveikslėlių perdavimo algoritmą teko adaptuoti šiai platformai, užuot panaudojus jau turimą kodą, kaip kitų modulių atveju. Kadangi tuo metu su šia kalba turėjom mažiausiai patirties, neatkreipėm dėmesio į tai, kad failų perdavimo metu gauto atvaizdo paskutinis paketas (kuris 150x150 pikselių atvaizdo atveju sudaro 2,9KB vietoj 5KB skirtų) buvo įrašomas į failą visas, o ne tiek, kiek skelbia jo dydžio reikšmė, taip į failą įrašant apie 2KB *šiukšlių* (*trash bytes*) – liktinės informacijos atmintyje nuo priešpaskutinio paketo, dėl ko galutinis failo dydis buvo 90KB. *Windows Bitmap* toleruoja papildomą informaciją failo gale, nes failo pradžioje aprašoma, kiek duomenų nuo pradžios reikia nuskaityti. Tačiau, kitų klaidų ieškojimo ir šalinimo procese šis kodas buvo modifikuotas, skaitymo klaida rasta ir sutvarkyta. Atrodytų, sistema turėtų veikti puikiai, tačiau pridėjus naują paveiksliuką sistema staiga visisškai atsisakė veikti – algoritmas „išmesdavo“ (*throw*) išimties atvejį (*exception*) ir sustodavo.

Po ilgų testavimų, derinimų paaiškėjo, kad problema iškilo kaip tik dėl sutvarkyto atvaizdų perdavimo kodo. Nauji paveiksliukai tapo mažesnio dydžio nei senieji, ko pasėkoje atpažinimo algoritmas gaudavo kelių skirtingų dydžių atvaizdus palyginimui ir, bandydamas traktuoti mažesnį atvaizdą kaip didesnio dydžio bandydavo skaityti už masyvo, saugančio atvaizdo duomenis, ribų. Pakeitus failų skaitymo algoritmą, kad skaitytų per tarpinį atvaizdo objektą algoritmo ir sistemos darbas normalizavosi.

5.1.8. Sistemos testavimas

Kai jau visi komponentai baigti ir sujungti, ateina laikas testuoti visą sistemą. Sistema testuojama pagal dokumentuotą funkcionalumą (juodos dėžės testavimas). Testavimas gali būti kaip automatizuotas, taip ir neautomatizuotas, tačiau automatizuotas testavimas retai panaudoja

komercinius įrankius, dėl kiekvienos sukurtos sistemos specifikų. Dažniausiai automatizuotam testavimui pasitelkiami pačių kūrėjų paruošti įrankiai.

VAĮKS atveju sistemos testavimas buvo neautomatizuotas. Sujungus sistemą į duomenų bazę buvo įtraukti sistemos kūrėjų, jų kolegų ar šiaip tuo metu netoli darbo vietos buvusių ir sutikusių pagelbėti žmonių atvaizdai (per administratoriaus sąsają). Įjungus sistemą paeiliui prie kameros prieidavo žmonės taip, kad kamera fiksuotų jų veido atvaizdą ir buvo stebima sistemos reakcija: atpažino ar neatpažino, įleido ar neįleido. Labai svarbų vaidmenį sistemos testavime atlieka iš anksto į programos kodą įtraukti informaciniai pranešimai apie programos būsenos pokyčius, veiksmus ir pan. Pagal dizainą šie pranešimai buvo sumanyti, kaip sistemos žurnalo dalis, skirta stebėti sistemos darbą ir diagnozuoti galimus trikčius. Tai sistemos *kūrimo testuojamumui* metodikos dalis. Kartu tame pačiame lange su būsenos informacija sistema atvaizduoja ir klaidų, jei tokios iškyla, informaciją.

5.2. Testavimo metodikų taikymo paskirstytų sistemų testavime tyrimas

Šiame skyrelyje apžvelgsiu analizės dalyje aprašytų testavimo metodikų panaudojamumą testuojant paskirstytas sistemas, įvardinsiu galimas panaudojimo sritis ir kokiose programos kūrimo stadijose šios metodikos gali būti taikomos.

5.2.1. Testavimo metodų vertinimo kriterijai (metrikos)

Sistemos kūrimo ir testavimo etapai:

- Sistemos dizainas:
  - Reikalavimų surinkimas;
  - Technologijų parinkimas;
  - Architektūros sprendimas;
  - Komponentų dizainas;
  - Komponentų sąsajų aprašymas;
- Sistemos kūrimas:
  - Komponentų kūrimas;
  - Sąsajų kūrimas;
  - Komponentų integravimas;
- Sistemos testavimas:
  - Klasių ir objektų testavimas;
  - Komponentų testavimas;
  - Sąsajų testavimas;

- Integracinis testavimas;
- Sistemos testavimas.

Testavimo metodikų vertinimo kriterijai:

- Testų kodo rašymo ir kompiliavimo reikiamumas
  Ar reikia taikant šia testavimo metodiką generuoti testų kodą ir jį kompiliuoti prieš vykdant/testuojant.

- Tinklo gedimų įtaka testams
  Ar testams daro įtaką tinklo gedimai – tinklo gedimas gali sukelti klaidingus testo rezultatus ar testas gali išvis nevykti.

- Pakartotinas panaudojamumas
  Ar testą (ar testo dalį) galima pakartotinai panaudoti kur nors kitur.

- Testavimo rezultatų stebėjimas
  Vertinimas, kaip patogiai galima stebėti testavimo rezultatus atliekant/atlikus testą.

- Testavimo automatizavimas
  Įvertinama galimybė automatizuoti testavimo procesą.

- Sąsajos testavimo galimybė
  Ar testavimo metodika gali būti naudojama sąsajai testuoti.

5.2.2. Testavimo metodikų pritaikomumas ir įvertinimas

Akivaizdu, kad skirtingos testavimo metodikos turi skirtingas testavimo paskirtis ir taikymo sritis. Apžvelgsiu pagrindinių analizės dalyje aprašytų testavimo metodikų taikymo sritis paskirstytų sistemų testavime.

2 lentelė. ***Testavimo metodikų analizė paskirstytų sistemų testavimo kontekste.***

Metodika	Taikymo sritis paskirstytų sistemų testavime
Statinė kodo analizė	Komponentų kūrimo metu: • dažniausiai pasitaikančių programavimo klaidų paieška; • sintaksės klaidų paieška; • „pavojingų“ kalbos konstrukcijų analizė; • indeksavimo klaidų paieška.

Kolegų peržiūra (*peer review*)	Komponentų kūrimo metu: • pavojingų programavimo metodų ir dažniausių klaidų paieška; • alternatyvių, optimalesnių sprendimų paieška. Visuose testavimo etapuose: • klaidų analizė; • klaidų priežasčių paieška; • klaidų sprendimo būdų analizė.
Vienetų testavimas (*unit testing*)	Komponentų kūrimo metu: • metodų ir funkcijų teisingo veikimo užtikrinimas; • klaidų paieška metoduose; • kodo savybių (padengimas, šakų skaičius, ciklų skaičius ir pan.) analizė; Sistemos palaikymo ir plėtojimo metu: • regresiniai testai.
Juodos dėžės (funkcionalumo) testavimas	Komponentų kūrimo metu: • komponentų, klasių atitikimo reikalavimams užtikrinimas; • komponentų, klasių sąsajos testavimas. Komponentų, integracinio ir sistemos testavimo metu: • sąsajos testavimas; • atitikimo reikalavimams užtikrinimas; • panaudos atvejų patikrinimas; • sistemos funkcionalumo testavimas.
Baltos dėžės (kodo savybių) testavimas	Komponentų kūrimo metu: • komponentų, klasių, metodų analizė; • algoritmo veikimo analizė; • rezultatų tikslumo analizė; Komponentų, integracinio ir sistemos testavimo metu: • sistemos funkcionalumo testavimas; • algoritmų funkcionavimo testavimas; • komponentų funkcionavimo testavimas;

Objektų testavimas	Komponentų kūrimo metu: • klasių kintamųjų ir metodų testavimas; • klasių metodų rezultatų analizė; • klasių sąsajų testavimas;
Kamščiais ir valdikliais paremtas testavimas	Komponentų kūrimo ir testavimo metu: • sąsajų kūrimas ir testavimas; • komponentų tinklo funkcionalumo kūrimas ir testavimas; • gali būti naudojami kaip kodo pagrindas komponentams kurti; • taipogi naudojami testuojant komponentus, kurie priklausomi nuo kitų komponentų funkcionalumo, testavimui (nesant ar nenorit naudoti kitų komponentų); Sistemos integravimo ir testavimo metu: • komponentų sąsajos testavimas; • sistemos reakcijos į tinklo trikčius testavimas;

Taigi, kaip matome, viena testavimo metodika nepadengsime visos programos ir tikrai nerasime visų klaidų. Kai kurios klaidos gali išlikti net nuodugniai ištestavus sistemą visomis įmanomomis metodikomis. Ir kuo didesnė bei sudėtingesnė sistema, tuo daugiau tokių pasleptų klaidų joje lieka. Svarbiausia testuojant suderinti skirtingas metodikas, tik taip galima pasiekti maksimalų rastų klaidų skaičių. Dauguma testavimo metodikų puikiai dera tarpusavyje, pavyzdžiui juodos dėžės testavimas komponentui puikiausiai dera su kamščių ir valdiklių taikymu. Kai kuriais atvejais šias metodikas net neįmanoma taikyti atskirai, ypač tai pasireiškia paskirstytose sistemose, kuriose vieni komponentai priklausomi nuo kitų.

Apžvelgus paskirstytų sistemų ir jų testavimo ypatumus galima drąsiai teigti, jog kamščių ir valdiklių taikymas yra labai svarbus norint sukurti patikimą, teisingai veikiančią paskirstytą sistemą. O pasirinkimas, kaip plačiai naudoti juos priklauso tik nuo programuotojo ir testuotojo norų bei patirties.

5.2.3. Testavimo metodikų palyginimas

3 lentelė. ***Testavimo metodikų taikymo paskirstytų sistemų testavimui palyginimas.***

Kriterijus	Statinė kodo analizė	Kolegų peržiūra	Vienetų testavimas	Juodos dėžės (funkcionalumo) testavimas	Baltos dėžės (kodo savybių) testavimas	Objektų testavimas	Kamščiais ir valdikliais paremtas testavimas
Testų kodo rašymas ir kompiliavimas	Nereikia	Nereikia	Reikia, dalis automatizuota	Reikia, galimas automatizavimas	Reikia, galimas automatizavimas	Reikia, galimas automatizavimas	Reikia, mažas automatizavimas
Tinklo gedimų įtaka testams	Jokios	Jokios	Jokios	Priklauso nuo testuojamo komponento	Priklauso nuo testuojamo komponento	Priklauso nuo testuojamo objekto	Didelė, jei testuojamas tinklo komponentas
Pakartotinas panaudojamumas	Nepanaudotinas	Nepanaudotinas	Regresijos testavimas	Regresijos testavimas, reikalavimų verifikavimas	Reikalavimų, algoritmų patikrinimas, profiliavimas	Objekto sąsajos testavimas	Galimas panaudojimas naujiems komponentams kurti
Testavimo rezultatų stebėjimas	Lengvas	Priklauso nuo pateikimo būdo	Lengvas	Lengvas	Lengvas	Priklauso nuo realizavimo	Priklauso nuo realizavimo
Testavimo automatizavimas	Visiškas	Jokio	Labai didelis	Didelis	Didelis	Vidutinis	Vidutinis
Sąsajos testavimo galimybė	Jokios	Teorinė	Labai maža	Vidutinė	Maža	Labai gera, bet tik objektams	Labai gera

Įvertinus skirtingas testavimo metodikas neįmanoma nustatyti, kuri iš jų yra geriausia, nes jų taikymo sritys programų/sistemų testavime yra skirtingos. Tačiau iš rezultatų galima matyti, kad daugelis testų reikalauja paruošimo rankiniu būdu (t.y., kad programuotojas pats paruoštų ir parašytų testų kodą), tačiau gali būti atliekami automatiškai. Daugelį testų reikia prieš naudojant sukompiliuoti (arba į atskirą programą/komponentą, arba kaip dalį testuojamos programos). Tinklo gedimai labiausiai įtakoja kamščiais paremtus tinklo komponentų ir sąsajų testavimą, nes šis testavimas reikalauja tvarkingai veikiančio tinklo. Tačiau kamščių testavimo metodika gali tyčia sukelti ar simuliuoti tinklo gedimus testavimo tikslais.

Smulkių sistemos komponentų testavimo metodikos (vienetų testai, juodos ar baltos dėžės, objektų testavimas) neturi platesnės pakartotino panaudojimo galimybės, apart taikymo regresiniame testavime. Statinė kodo analizė ir kolegų peržiūra, kadangi pati negeneruoja testavimo kodo, negali būti taikoma pakartotiname panaudojime. Lengviausiai pakartotinai panaudojamas testavimo metodikos produktas yra kamščių ir valdiklių kodas, kuri galima pritaikyti naujų komponentų kūrime jų pagrindu ar jų kodo dalių pritaikymu komponentų funkcionalumui išplėsti.

Rezultatų stebėjimas visoms testavimo metodikoms yra nesudėtingas, tačiau neretai smarkiai priklauso nuo testavimo metodikos realizavimo ir naudojamų įrankių.

Sąsajoms testuoti geriausiai tinka objektinis testavimas (jei testuojama objekto ar klasės sąsaja) ir kamščiais bei valdikliais paremtas testavimas. Valdikliai ir kamščiai geriausiai taikomi būtent testuojant komponentų sąsajas, nes tokios galimybės neturi kitos apžvelgtos testavimo metodikos.

6. Išvados

- Darbo metu išanalizuotos paskirstytų sistemų architektūros, jų ypatumai, privalumai ir trūkumai. Apžvelgtos populiariausios testavimo metodikos.
- Mes susiduriame su paskirstytomis sistemomis bene visur, daugiausia jų sutinkama internete, nes sparčiai paplitus ir išpopuliarėjus internetui vis daugiau programų kūrėjų stengiasi pasinaudoti jo teikiamais privalumais.
- Net ir paprasčiausią programą, kuri naudojasi nutolusio serverio paslaugomis galima vadinti paskirstytos sistemos dalis, nes serveris fiziškai nėra tame pačiame kompiuteryje o gali būti kad ir kitoje pasaulio pusėje.
- Tačiau taip plačiai išplitus paskirstytoms sistemoms vis dar nėra konkrečių testavimo metodikų, skirtų testuoti paskirstytas sistemas.
- Plačiausiai paskirstytų sistemų testavime taikoma valdiklių ir kamščių testavimo metodika, tačiau ji yra derinama su kitomis metodikomis, neretai neturinčių nieko bendro su paskirstytomis sistemomis.
- Paskirstytų sistemų kūrimas ir testavimas yra labai sudėtingas ir atsakingas procesas, kadangi paskirstytų sistemų architektūra yra sudėtingesnė už lokalių ir yra žymiai daugiau faktorių, neretai net nepriklausančių nuo programos kūrėjų (tinklas, tinklo komponentai, sistemų komponentai ir t.t.), kurie gali įtakoti sistemos darbą ir į jų poveikį reikia atsižvelgti.
- Kuriant paskirstytas sistemas privalu laikytis požiūrio, kad jei klaida gali nutikti, ji būtinai nutiks. Todėl reikia sistemą parengti taip, kad tokie gedimai arba neįvyktų, arba jei įvyktų, sudarytų kuo mažiau problemų sistemos funkcionavimui.
- Labai naudinga yra kuriant tiek paskirstytas, tiek lokalias sistemas laikytis *kūrimo testuojamumui* metodikos. Laikantis šios metodikos supaprastės tolimesnis sistemos testavimas.
- Svarbiausia, testuojant tiek paskirstytas, tiek nepaskirstytas sistemas nėra vienos absoliučiai geriausios testavimo metodikos. Tik derinant skirtingas metodikas galima pasiekti gerą sistemos ištestavimą ir minimizuoti klaidų skaičių.

7. Literatūros sąrašas

1. Introduction to Distributed System Design - Google Code University - Google Code - http://code.google.com/edu/parallel/dsd-tutorial.html – žiūrėta 2010.04.10
2. Distributed computing - Wikipedia, the free encyclopedia - http://en.wikipedia.org/wiki/Distributed_systems – žiūrėta 2010.04.10
3. History of distributed Systems, Joseph Cordina – http://www.cs.um.edu.mt/~jcord/Historical.pdf – žiūrėta 2010.04.25
4. Symposium on Principles of Distributed Computing - Wikipedia, the free encyclopedia – http://en.wikipedia.org/wiki/Symposium_on_Principles_of_Distributed_Computing – žiūrėta 2010.04.25
5. International Symposium on Distributed Computing - Wikipedia, the free encyclopedia – http://en.wikipedia.org/wiki/International_Symposium_on_Distributed_Computing – žiūrėta 2010.04.25
6. Designing Distributed Systems, A Conversation with Ken Arnold, Part III – http://www.artima.com/intv/distrib.html – žiūrėta 2010.04.17
7. Unusual software bug - Wikipedia, the free encyclopedia – http://en.wikipedia.org/wiki/Heisenbug – žiūrėta 2010.04.12
8. Birman, Kenneth. Reliable Distributed Systems: Technologies, Web Services and Applications. New York: Springer-Verlag, 2005.
9. Rutgers University: Distributed Computing, Google Code University - http://code.google.com/edu/submissions/rutgers/index.html - žiūrėta 2010.04.26
10. P2P – Vikipedija – http://lt.wikipedia.org/wiki/P2P – žiūrėta 2010.04.26
11. Peer-to-peer – Wikipedia – http://en.wikipedia.org/wiki/Peer-to-peer – žiūrėta 2010.04.14
12. Nathan Boy, Jared Casper, Carlos Pacheco, Amy Williams - Automated Testing of Distributed Systems – http://people.csail.mit.edu/cpacheco/publications/BoyCPW2003.pdf – žiūrėta 2010.04.10
13. University of Colorado, Boulder – Software Engineering of Distributed Systems: Testing Distributed Software – http://ecee.colorado.edu/~swengctf/distributed/presentations/ECEN5053Wk13TestingDistSW.ppt – žiūrėta 2010.04.04
14. Synchronization (computer science) - Wikipedia, the free encyclopedia – http://en.wikipedia.org/wiki/Synchronization_(computer_science) – žiūrėta 2010.05.01

15. Design for Testability, Steven r. Rakitin, software Quality Consulting Inc. – http://www.swqual.com/SQGNE/presentations/2006-07/Rakitin%20Oct%202006.pdf – žiūrėta 2010.05.01
16. Food for Thought: Design for Testability – http://www.swqual.com/newsletter/vol3/no3/vol3no3.html - žiūrėta 2010.05.01
17. Client–server model - Wikipedia, the free encyclopedia – http://en.wikipedia.org/wiki/Client-server – žiūrėta 2010.05.14
18. Middleware - Wikipedia, the free encyclopedia – http://en.wikipedia.org/wiki/Middleware – žiūrėta 2010.05.14
19. Ex-Sight.Com Advanced Biometrics. – www.ex-sight.com – žiūrėta 2009.10.22
20. NEC's NeoFace Control Security and Identification System of the Highest Level. – http://www.nec.com.hk/english/press/press_detail.php?id=184 – žiūrėta 2009.10.22
21. RCG. FxGuard Pro Biometric Access Control with Face Recognition Technology.– http://www.rcgstore.tv/WebLITE/Applications/productcatalog/uploaded/Docs/FxGuard%20Pro_.pdf – žiūrėta 2009-10-22