VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
PROGRAMŲ SISTEMŲ STUDIJŲ PROGRAMA

Programų sistemų našumo sprendimų išreikštų Aktorių modeliu įgyvendinimas

Software Systems Efficiency Solutions Expressed On The Actors Model Implementation

Magistro baigiamasis darbas

Atliko: Aurimas Kraulaidys (parašas)

Darbo vadovas: Donatas Čiukšys (parašas)

Recenzentas: doc. dr. Audronė Lupeikienė (parašas)

Vilnius – 2021

SANTRAUKA

Kuriant šiuolaikines programų sistemas nebeužtenka skirti dėmesį vien tinkamos programavimo kalbos, atitinkamų karkasų ar architektūros pasirinkimui. Nuolatos didėjant duomenų kiekiams, kasmet vis svarbesnę sistemos projektavimo dalį užima kuriamos sistemos našumo sprendimai, kurie leistų sistemai veikti efektyviai net duomenų kiekiams ir toliau augant. Šiame magistro darbe nagrinėjamas programų sistemų našumo sprendimų išreikštų aktorių modeliu įgyvendinimas.

SUMMARY

Developing modern software systems is not enough to choose a proper programming language, suitable frameworks and architecture. The amount of data is constantly growing as a result significant part of system design is dedicated to it‘s efficiency solutions, so that system would still keep working in a fast and stable mode. In this Master thesis we analyze software systems efficienty solutions expressed on the actor model.

PROBLEMA

Nėra metodinių patarimų kaip įgyvendinti programų sistemos našumo sprendimus remiantis Aktorių modeliu.

DARBO TIKSLAS

Šios magistro darbo dalies tikslas rasti būdus kaip identifikuoti Aktorius, sistemos reikalavimų įgyvendinimui.

UŽDAVINIAI

1. Turint panaudojimo atvejus, rasti būdus, kuriais remiantis būtų galima identifikuoti Aktorius;
2. Aprašyti galimas pranešimų tarp Aktorių paskirstymo strategijas;

Turinys

ĮVADAS

Pastaraisiais dešimtmečiais drastiškai keičiantis verslo aplinkoms, kurioms valdyti ir spręsti kylančius uždavinius yra kuriama galybė įvairiausios paskirties programinės įrangos, augant vartotojų lūkesčiams, o sistemoms atskirais atvejais baigiant pasiekti savo galimybių ribas, pradeda aiškėti, kad ateities programų sistemos nebegalės būti kuriamos taip, kaip tai buvo daroma pastaruosius dešimtmečius. Spartus duomenų ir vartotojų kiekio augimas, bei praktiškai pasiektos procesorių galimybių ribos - pagrindiniai faktoriai verčiantys ieškoti būdų, o kaip turėtų būti kuriamos programų sistemos susiduriančios su šiais iššūkiais.

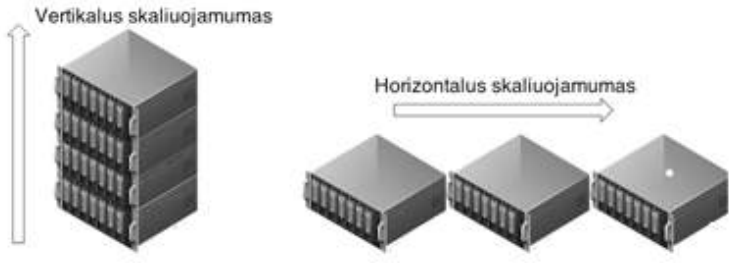
1. FAKTORIAI VERČIANTYS KEISTI POŽIŪRĮ Į PROGRAMŲ SISTEMŲ KŪRIMO BŪDUS

1.1. Augantys duomenų ir vartotojų kiekiai

Duomenų kiekiams didėjant milžinišku greičiu iš programų sistemų kasmet tikimasi ne mažesnio stabilumo, didesnio atsakymo greičio, lankstumo ir patikimumo. [BMOYTKWB13] Jei dar prieš dešimtmetį aplikacijos tenkinosi dešimtimis serverių, jų reakcijos laikas buvo skaičiuojamas sekundėmis, tolerancija nepasiekiamumui realiu laiku siekdavo valandas, o duomenų kiekiai buvo skaičiuojamai gigabaitais, šiais laikais vartotojai tikisi reakcijos laiko milisekundėmis, 100% pasiekiamumo 24/7, o duomenų kiekiai skaičiuojami petabaitais. Tuo tarpu globalizacijai ir interneto galimybėms pasiekiant vis naujas aukštumas matome, tokias aplikacijos kaip Facebook ar Twitter, kurių aktyvūs vartotojai skaičiuojami šimtais milijonų (pvz. Facebook 1,5 milijardo). Panašu, kad ateityje programų sistemų, turinčių didelius kiekius vartotojų tik daugės, be to sistemos turėtų būti elastingos - gebėti prisitaikyti prie didėjančių ar mažėjančių apimčių.

Išaugęs sistemos apkrovimas fiziškai pridėjus daugiau resursų gali būti valdomas dviem būdais:

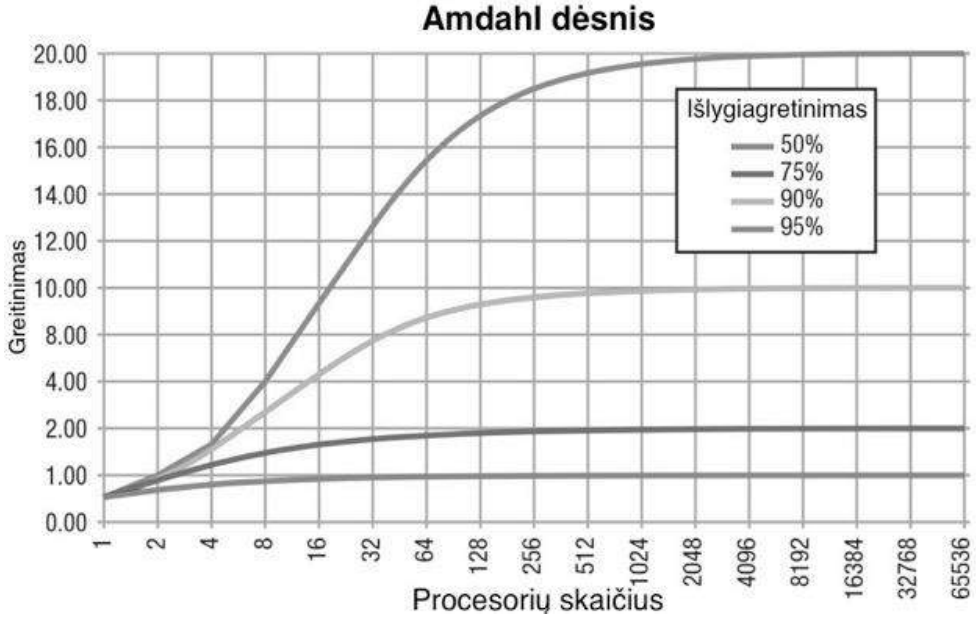
- didinant konkretaus kompiuterio pajėgumus – **vertikalus skaliuojamumas** (*angl. scale up*), pvz. pridedant daugiau procesorių vienam kompiuteriui (serveriui) arba
- apjungiant kelis kompiuterius (serverius) į klasterius (*angl. cluster*), kurie vartotojo požiūriu veikia kaip vienas įrenginys – **horizontalus skaliuojamumas** (*angl. scale out*).



1 pav. Vertikalus ir horizontalus skaliuojamumas

1.2. Procesorių galimybės

Ilgą laiką buvo galima stebėti tendenciją, kai kas 2-3 metus procesorių greičiai bemaž padvigubėdavo, tai leisdavo susidoroti su augančiomis apimtimis ir iš to kylančiais iššūkiais, tačiau procesorių pajėgumui praktiškai pasiekus fizikinių galimybių ribas, nelieka nieko kito, kaip tik apjungti daugelį procesorių ir dirbti su jais kaip vienu junginiu, taip maksimizuojant tokio darinio galimybes. Tai ypač tampa aktualu, kai reikia apdoroti didelius duomenų ir vartotojų kiekius. Tam, kad užtikrinti kaip įmanoma spartesnį programų veikimą, siekiama maksimalaus jų vykdymo išlygiagretinimo - kai atskiri procesoriai vykdo tam tikrą programos dalį, žymiai paspartindami uždavinio sprendimą, žinoma, jei tik jis savo esme gali būti išlygiagretinamas. Deja, ne viskas taip paprasta. Sprendžiant išlygiagretinimo ir spartinimo problemas yra žinomas Amdahl dėsnis[5], kuris teigia, kad vien tik papildomo procesorių kiekio pridėjimas nepadės išspręsti spartinimo problemos, pvz. pasiekus programos išlygiagretinimą net iki 95 proc. - spartinimas padidės tik iki 20 kartų, nesvarbu kiek daug procesorių bus naudojama (2 pav.). Tai pasako vieną paprastą dalyką - procesorių kiekis yra svarbus, tačiau nepakankamas faktorius norint pasiekti pageidaujamą spartą, neužtenka vien praplėsti aparatūros galimybių, tam reikia ieškoti programinių sprendimų.



2 pav. Amdahl dėsnis [SJ13].

1.3. Sudėtingas lygiagretaus programavimo sprendimų įgyvendinimas

Pažvelgę giliau į programinius sprendimus pamatysime, kad nėra viskas taip trivialu. Lygiagretus programavimas reikalauja itin aukštos kvalifikacijos programuotojų, bet kokios klaidos ne tik, kad gali nepasiekti pageidaujamo spartinimo, priešingai, jį gali sumažinti, be to, programuotojo klaida gali sukelti nedeterministinius rezultatus, kas nėra jau toks retas atvejis programuojant lygiagrečiai.

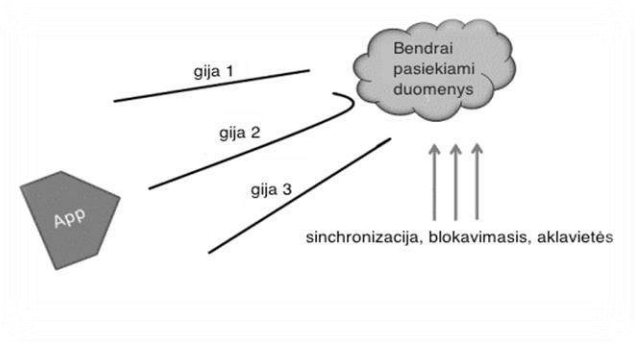
Lygiagrečiai programuojant klasikiniu būdu, neišvengsime sudėtingo ir painaus programinio kodo, reikalingos gilios žinios ir didelė patirtis išlygiagretinimo srityje, kaip minėta, dažnai paliekama klaidų. Iš to peršasi išvada, jog negerai išlygiagretinimo klausimą palikti programuotojams. [McK15]

1.4. Klasikinių technologijų trūkumai ir alternatyvos

Matant poreikius, kurie kyla iš aukščiau aprašytų situacijų pavyzdžių, peršasi išvada, jog populiarios, klasikinėmis vadinamos, daugelio lygmenų (*angl. n-tier*) sistemos susiduria su iššūkiais (3 pav.), kai patenkinti nūdienos reikalavimus darosi ne tik, kad labai brangu, tačiau kartais ir sunkiai įmanoma, todėl reikia ieškoti kitų sprendimų.

Norėtume išskirti šiuos mūsų nuomone pagrindinius klasikinių technologijų trūkumus išlygiagretinimo kontekste :

1. Sistemos elementai dažniausiai komunikuoja sinchroniškai;
2. Sinchroninis komunikavimas sukelia blokavimąsi.
3. Išlygiagretinimo projektavimo ir įgyvendinimo klausimai paliekami programuotojams.



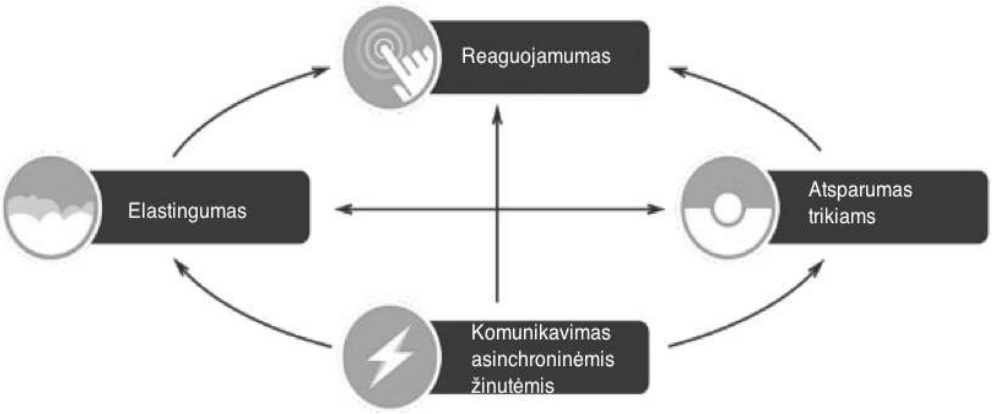
3 pav. Klasikinių technologijų lygiagretaus programavimo iššūkiai

Pastariesiems dviem trūkumams spręsti galima būtų išnaudoti tokias technologijas kaip programų sistemų transakcijų atmintis (*ang Software Tranzactional Memory(STM)*), duomenų

srauto išlygiagretinimas (*ang. DataFlow Concurency*), Aktorių modelis (*angl. Actors*). Deja, jos nėra pilnai pritaikytos klasikinėms programų sistemų architektūroms ir nors teorinių modelių pavyzdžių literatūroje galima rasti, praktikoje sprendimai įgyvendinami vangiai, ko viena iš priežasčių galėtų būti informacijos ir metodologinių patarimų stoka, kaip teorinius modelius susieti ir įgyvendinti šiuolaikinėmis priemonėmis, kurios leistų programuotojui nesirūpinti sudėtingu ir iš to kylančia išvada, pakankamai brangiu išlygiagretinimo sprendimų įgyvendinimu, tačiau įgalintų naudotis žinomais išlygiagretinimo mechanizmais, kurie palengvintų spręsti anksčiau darbe paminėtas problemas. Kaip vienas iš sprendimų būdų galėtų būti programų sistemos, paremtos Aktorių modelių, tačiau šiai dienai nėra pateikiama metodinių nurodymų, kaip tipinius išlygiagretinimo ir kitus sistemos našumą didinančius sprendimus įgyvendinti Aktorių modeliu, pavyzdžiui, kiek panaudojimo atvejui įgyvendinti reikės Aktorių, kaip tie Aktoriai komunikuos, koks bus jų gyvavimo ciklas ir daug kitų klausimų, kurie iškyla teorinį modelį norint įgyvendinti praktiškai. Aktorių modelį ir jo galimybes įgyvendinant programų sistemų našumo sprendimus nagrinėsime šiame magistro darbe.

1.5. Reaguojančios sistemos

Programuotojų pasaulis mato, kad spręsti visas aukščiau paminėtas problemas reikia sprendimo, o ieškant problemos sprendimo būdų vis dažniau šiuolaikinėje literatūroje galime galime sutikti terminą - reaguojančios sistemos. Vieni tai vadina filosofija, kiti savybių rinkiniu, kuriomis turi pasižymėti šiuolaikinė programų sistema. Nors pats terminas „reaguojanti sistema“ nėra senas, tačiau filosofija ir sprendimai, kurie slypi po šiuo pavadinimu nėra nauji. Sprendimai buvo žinomi jau anksčiau - aplikacijos, turinčios didelius kiekius vartotojų, visais laikais buvo priverstos ieškoti sprendimo būdų, kas skatino kurti ir naudoti savo poreikiams kažką panašaus, kas gali būti įvardijama kaip reaguojanti sistema. Duomenų ir vartotojų kiekiui toliau augant, o procesoriams beveik pasiekus savo galimybių ribas šie būdai prisiminti iš naujo ir dabar yra galbūt aktualesni kur kas didesniam kiekiui programų sistemų, nei tuo momentu, kai buvo išrasti. Tyrėjų nuomone [VER15] [HP85] [BMOYTKWB13] reaguojančios sistemos gali padėti spręsti darbe nagrinėjamas problemas.



4 pav. Reaguojančių sistemų savybės.

Reaguojančių sistemų šalininkai [BMOYTKWB13] skelbia, kad šiuolaikinė reaguojančių sistemų programinė įranga turi pasižymėti keturiomis pamatinėmis savybėmis, kurios tarpusavyje tampriai susijusios (4 pav). Taigi reaguojančios sistemos turi būti:

- Elastingos;
- Reaguojančios;
- Atsparios trikiams;
- Komunikuojančios asinchroninėmis žinutėmis;

Aptarkime detaliau, kas slypi po šiais pavadinimais.

- **Elastingos** (*angl. elastic*). Turi būti galimybė sistemai reaguoti į didėjančias ar mažėjančias aplikacijų apkrovas (duomenų kiekio, vartotojų skaičiaus pasikeitimai), pavyzdžiui staiga ar dėl sezoninių svyravimų išaugus operacijų skaičiui sistema turi galėti plėstis (naujų serverių prijungimas ir kitokio tipo fizinio galingumo didinimas). Lygiai taip pat pasikeitus poreikiams turi būti galimybė pajėgumus mažinti. Idėja, kad resursai, kurių reikia būtų įdarbinami esant konkrečiam poreikiui, o jei poreikio nėra resursai atlaisvinami. Taip pigiau, nereikia pirkti visos įrangos iškart nežinant ar tikrai ja pasinaudosi, nusipirkus per daug bus prastovos, neturint pakankamai įrangos yra ribojamos galimybės ir vienu ir kitu atveju patiriami nuostoliui.
- **Reaguojančios** (*angl. responsive*). Turi būti užtikrinamas greitas atsako laikas į stimulą.
- **Atsparios trikiams** (*angl. resiliant*) t.y. gebančios greit atsigauti. Visos sistemos genda, reaguojanti sistema turi pasižymėti savybe, kad netgi įvykus tam tikriems sistemos sutrikimams ji toliau funkcionuotų, o gedimai butų pašalinami pačios sistemos.

- **Komunikuojančios asinchroninėmis žinutėmis** *(angl. message driven).* Reaguojančios sistemos elementai komunikuoja asinchroniškai keičiantis žinutėmis. Komponentai nenaudoja kitų komponentų vidinių duomenų struktūrų, nėra blokavimosi, todėl išsiuntus žinutę siuntėjas neturi laukti atsakymo, o kol jį gaus gali atlikinėti kitas užduotis, taip suteikiant sistemai efektyvumo.

Kaip vėliau pastebėsime šiame darbe, reaktyvių sistemų savybės gali būti pasiekiamos sprendimus įgyvendinant Aktorių modeliu. Aktorių modelis, taip pat nėra naujovė, tačiau jo teikiamos galimybės ypač aktualios sprendžiant šiuolaikinių dideles apkrovas patiriančių programų sistemų problemas. Aktorius formaliai yra fundamentalus skaičiavimo vienetas [HMS12], veikiantis Aktorių sistemoje. Keičiantis resursų kiekiems Aktorių modelis leidžia pasiekti *Elastingumą*, dėl galimybės pridėti (sumažinti) darbą atliekančių Aktorių skaičių, kurie gali būti tiek lokalioje mašinoje tiek nutolusiuose taškuose (pvz.: serveriuose). *Greitas atsako laikas* gali būti pasiekiamas išnaudojant lygiagretaus veikimo sprendimus, kurie yra viena iš Aktorių sistemos savybių. *Atsparumas trikiams* gali būti užtikrinamas priskiriant specialiai dedikuotus Aktorius, kurie atsakingi tam tikrą programinio kodo bloko veikimo priežiūrą ir veiksmus įvykus įvykus sutrikimams. Viena iš kertinių Aktoriaus savybių – tarpusavyje Aktoriai *komunikuoja asinchroninėmis žinutėmis*. Prie šių savybių dar grįšime ir jas išanalizuosime tolesniuose šio darbo skyriuose, o dabar detaliau panagrinėkime patį Aktorių modelį.

2. AKTORIŲ MODELIS

Aktorių modelis pirmą kartą buvo pristatytas 1973 metais [HBS73]. Modelio autoriai - Carl Hewitt, Perter Bishop, Richard Steiger. Vėliau tokie autoriai kaip Henry Baker [HB78], Gul Agha [AGH86] [KA11] ir kiti prisidėjo prie Aktorių modelio teorijos vystymo. Šis modelis pateikia lanksčius mechanizmus išlygiagretinimo sprendimais paremtų programų sistemų kūrimui. Aktorių modelis nuo kitų išlygiagretinimo modelių skiriasi visų pirmą tuo, kad jis paremtas asinchroniniu pranešimų keitimusi. Aktorių modelio esminė esybė yra Aktoriaus - subjektas, kuris ir nusako, kaip skaičiavimai turi būti atliekami. [HAL10]

Pagrindinė modelio idėja yra, kad elementai, kuriais operuojama nekeičia būsenos (*angl. imutable state*), elementai nėra pasiekiami tiesiogiai, o darbų paskirstymui ir koordinavimui tarp Aktorių naudojamas asinchroninis pranešimų siuntimas (*angl. message-passing*). [M97] [HEW15]

Aktorių modelis paremtas filosofija, kad viskas yra Aktorius, panašiai kaip objekto filosofija, kuria vadovaujamasi daugelyje objektinių programavimo kalbų, sutariant, kad viskas yra objektai, tačiau šios dvi filosofijos skiriasi visų pirma tuo, kad objektine paradigma paremta programų sistema dažnu atveju vykdoma nuosekliai, tuo tarpu Aktorių modelio veikimas prigimtinai lygiagretus. Išaugęs lygiagretaus programavimo poreikis, dėl tokių priežasčių, kaip plačiai naudojamos debesų kompiuterijos galimybės, taip pat programavimas daugelio branduolių aplinkose sudarė sąlygas iš naujo prisiminti Carl Hewitt su kompanija pasiūlytą koncepciją, bei remiantis Aktorių modelio idėja intensyviai ieškoti galimybių šiuolaikiniams uždaviniams spręsti.

2.1. Klasikinis Aktorių modelis

Pats idėjos autorius Carl Hewitt naujausioje savo straipsnio „Actor Model of Computation: Scalable Robust Information Systems“ (2015) publikacijoje, Aktorių apibrėžia, kaip skaičiavimo subjektą, kuris, atsakydamas į gautą pranešimą, lygiagrečiai gali [H15]:

- siųsti žinutes kitiems Aktoriams adresais, kuriuos jis turi;
- sukurti naujus Aktorius;
- nuspręsti kaip elgsis gavęs kitą žinutę.

Šių veiksmų atlikimas neturi eiliškumo ar pirmenybės vienas kito atžvilgiu ir gali būti atliekami lygiagrečiai, be to išsiųstos žinutės gali būti gautos, bet kokia eilės tvarka. Tiesioginio ryšio tarp

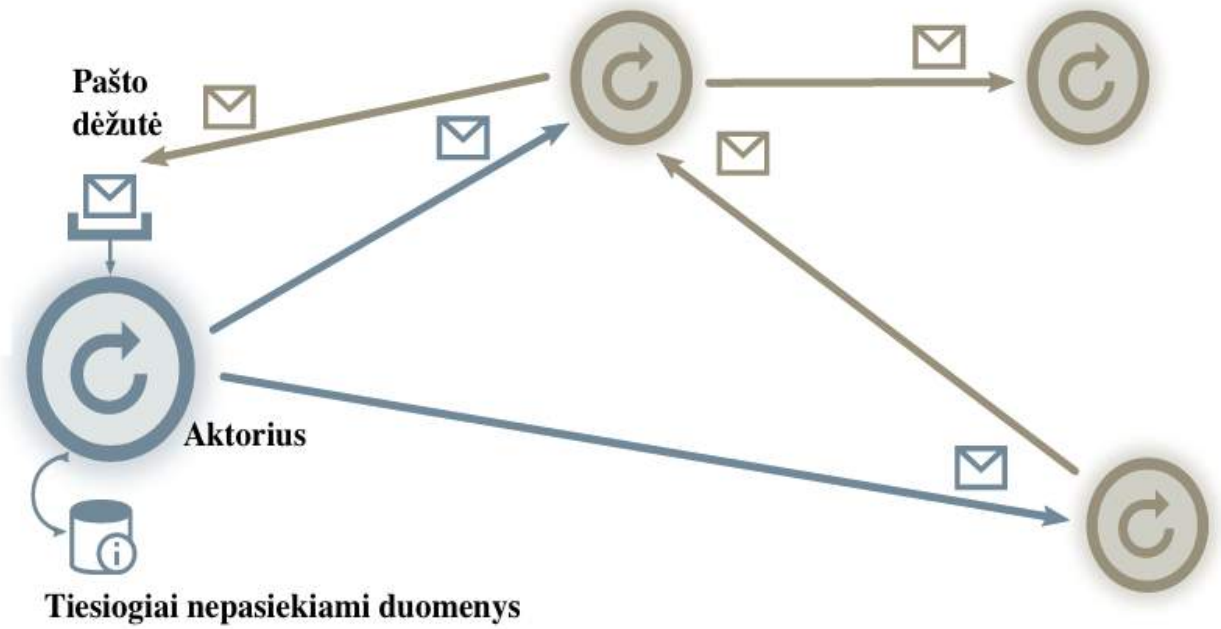
siuntėjo ir gavėjo eliminavimas tapo fundamentalia Aktorių modelio savybe. Ši savybė turėjo reikšmingą poveikį asinchroninio komunikavimo galimybių vystymui, bei turėjo įtakos tokių modelių plėtojimui, kaip pavyzdžiui Žinučių Siuntimo (*angl. Message-passing*) modelis.

Supaprastinant Aktoriaus apibrėžimą Aktorius gali būti nusakomas kaip objektas, kuris geba apdoroti gaunamus pranešimus, o taip pat geba tokius pranešimus siųsti. Kiekvienas Aktorius turi apibrėžtą gyvavimo ciklą ir apribojimus savo veiksmams priklausomai nuo to, kokiame gyvavimo ciklo etate yra esybė [ShS12]. Detalesni Aktorių modelio, jo elementų, tame tarpe ir paties Aktoriaus kaip esybės apibrėžimai pateikti šio darbo skyriuje „Aktorių modelio sąvokų sistema“.

Aktorių modelio kontekste, pranešimų gavėjai yra identifikuojami pagal adresą, kartais literatūroje galima sutikti jį vadinant pašto adresu (*angl. Mailing address*), taigi, kad Aktorius galėtų komunikuoti su kitu Aktoriumi, jis turi žinoti jo adresą. Adresas gali būti gautas kartu žinute arba gali būti adresas Aktoriaus, kurį jis pats sukūrė.

Aktorių modelio išskirtinės savybės:

- žinutės apdorojamos lygiagrečiai;
- žinučių Siuntimas, gali būti laikomas tokiu pat paprastu ir greitu dariniu, kaip pavyzdžiui procedūros iškvietimas;
- Primityvūs Aktoriai gali būti įdiegti aparatūrinėje įrangoje (*angl. Hardware*).



5 pav. Aktorių modelis [ERB12]

2.2. Haller-Odersky Aktorių modelis

Nors Aktorių modelio koncepcija buvo pristatyta dar 1973 metais [HBS73], tačiau realus jo naudojamas programų sistemų kūrimo sprendimuose kiek užtruko. Kol globalių aplikacijų vartotojų skaičius nebuvo pasiekęs kritinio lygio, daugiau ar mažiau sėkmingai buvo sprendžiama tam momentui populiariais sprendimais, kurie reikalavo ir tebereikalauja programuotojui pačiam spręsti išlygiagretinimo sprendimus. Kita vertus tam, kad Aktorių modelis būtų plačiai pritaikomas ir lengvai implementuojamas, ilgą laiką buvo didelis atotrūkis tarp C.Hewitt, P. Bishop, R. Steiger, H. Baker, bei G. Agha vystyto Aktorių teorinio modelio [HBS73] [HB78] [AGH86], ir jo įgyvendinimo praktiškai veikiančiose aplikacijose.

Milžinišką indėlį šioje srityje įdėjo mokslininkai Philipp Haller bei Martin Odersky, kurie ne tik tyrinėjo ir vystė klasikiniu laikomą Aktorių modelį, tačiau stengėsi jį pateikti taip, kad jis būtų ne vien teorinė paradigma, tačiau būtų galima įgyvendinti praktiškai, realiai veikiančiose sistemose, bei būtų tinkamas realizuoti įprastomis objektinio programavimo kalbomis [HO06] [HO07] [HO08] [H10]. Aktorių modelį 2006 Philipp Haller realizavo kaip papildomą Scala programavimo kalbos galybę, o 2010 Jonas Bonér sukūrė AKKA karkasą, kuriame apjungė Scala ir Java kalbos galimybes, bei kuriame yra paruoštas naudoti Aktorių modelio mechanizmas.

Philipp Haller bei Martin Odersky pasiūlytas [HO08] [H10], bei AKKA platformos vardu pozicionuojamas Aktorių modelis, tai teorinio Carl Hewitt Aktorių modelio realizacija, kuris jau turi įgyvendintas klasikinio Aktorių modelio savybes, bei papildomą praplėstą modelio notaciją, pagrinde susijusią su modelio įgyvendinimu praktiškai veikiančiose sistemose. Kitame skyriuje pateikiama Aktorių modelio pagrindinių sąvokų sistema apimanti tiek, taip vadinamą, klasikinį Aktorių modelį tiek AKKA siūlomą notaciją.

2.3. Aktorių modelio sąvokų sistema

Šiame skyriuje apžvelgsime pagrindines sąvokas, sutinkamas Aktorių modelio kontekste, bei panagrinėsime, kada ir kaip pateiktos Aktorių modelio savybės yra naudojamos.

- **Aktorių sistema** (*angl. actor system*)

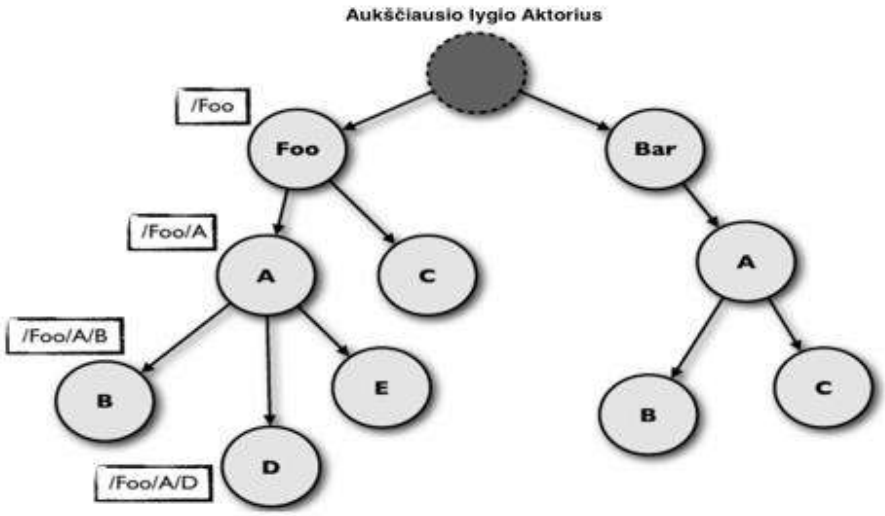
  Sistema, kurioje veikia Aktoriai. Šioje struktūroje gali būti vienas arba daugiau Aktorių, sujungtų į grupę. Paprastai Aktoriai į Aktorių sistemą, kaip struktūros gaubiantį elementą, yra sujungiami pagal joje veikiančių Aktorių uždavinio sprendimo pobūdį. Aktoriai egzistuojantys Aktorių sistemoje naudojasi bendrais resursais (pavyzdžiui

gijomis). Aplikacija, priklausomai nuo problemos sprendimo kompleksiškumo ir apdorojamų uždavinių, dalykinės srities homogeniškumo, gali turėti nuo vienos iki daug Aktorių sistemų. Kaip yra pasakęs pats Carl Hewitt [HMS12] vienas Aktorius yra ne Aktorius, turėdamas omeny, kad tik daug Aktorių veikdami Aktorių sistemoje gali pasiekti gerąsias Aktorių modelio savybes. Vienas Aktorius pats savyje neturi išlygiagretimo mechanizmo, tačiau Aktorių grupė veikdama sistemoje leidžia pasiekti paprastus ir gerai veikiančios sprendimus šioje srityje

- **Aktorių sistemos hierarchinė struktūra**

  Aktorių sistema yra hierarchiškai aukščiausias struktūros elementas. Aktorius gali kurti kitus Aktorius. Aktorius sukūręs kitą Aktorių tampa tėviniu Aktoriumi, sukurtojo Aktoriaus atžvilgiu, tuo tarpu sukurtasis tampa Aktoriumi - vaiku, šis savo ruožtu gali kurti savo vaikinius Aktorius

  Pavyzdžiui AKKA Aktorių sistemoje sukurtas Aktorius įgyja vardą tiesioginiai pagal kelią hierarchinėje medžio struktūroje (6 pav.).



6 pav. AKKA Aktorių hierarchinė struktūra ir Aktoriaus pasiekiamumas sistemoje [M015].

- **Aktorius** (*angl. actor)*

  Tai mažiausias Aktorių sistemos elementas. Aktoriai turi būseną, jie nesiblokuoja ir su kitais

  Aktoriais komunikuoja asinchroniškai siųsdami vienas kitam pranešimus. Kad galėtų tarpusavy komunikuoti, Aktoriai turi žinoti vienas kito adresus.

  Aktorius gali:

- asinchroniškai siųsti pranešimus kitiems Aktoriams;
- sukurti kitą Aktorių ir jį paleisti, duodant jam užduotį;
- sustabdyti sukurtą Aktorių arba patį save;
- apdoroti gautą pranešimą pats arba jį perduoti kitam Aktoriui, priklausomai nuo dalykinės srities realizacijos ir konfigūravimo taisyklių.[HMS12]

- **Aktoriaus gyvavimo ciklas** (*angl. actor life cycle)*

  Per visą gyvavimo ciklą Aktorius gali patirti tris būsenas:
  - *Naujas* (*angl. new)* : Aktoriaus esybė yra sukurta. Šioje būsenoje Aktorius dar negali nei gauti nei apdoroti pranešimų;
  - *Pradėjęs darbą* (*angl. started):* Aktorius įgyja šią būsena po to, kai išreikštinai kviečiama operacija, nusakanti, kad sukurtas Aktorius turi būti pasirengęs pradėti darbą. Šioje būsenoje aktorius jau gali gauti asinchroninius pranešimus ir juos apdoroti. Apdorodamas gautus pranešimus, Aktorius nesiblokuoja ir yra visada pasirengęs priimti kitus pranešimus iš siuntėjų;
  - *Sustabdytas* (*angl. terminated*): Ši būsena reiškia, kad Aktorius nebegali nei gauti naujų žinučių, nei jų apdoroti, į ją pereinama po būsenos „pradėjęs darbą“.

- **Aktorius siuntėjas** (*angl. sender*)

  Aktorius, kuris siunčia pranešimą Aktoriui gavėjui. Šią abstrakcija Aktorius įgauna tik komunikacijos metu.

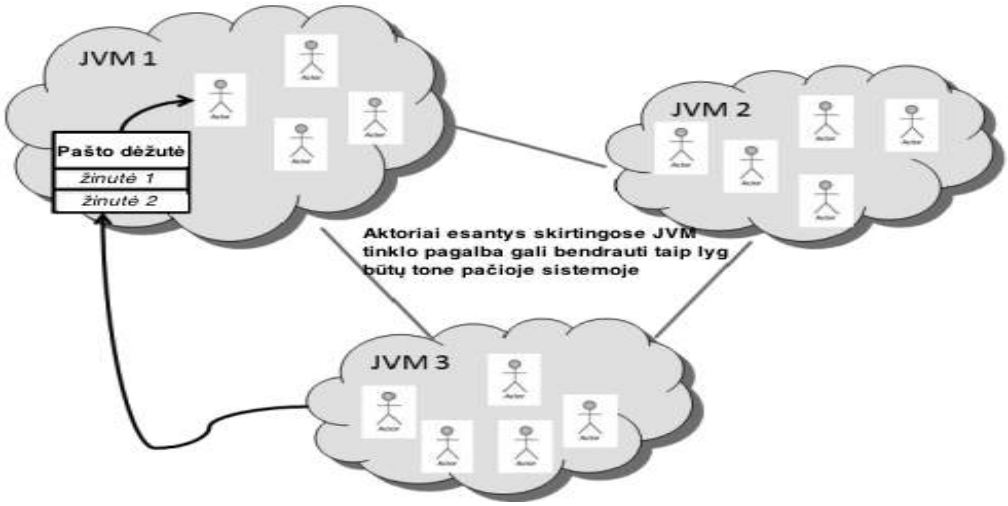
  Aktorius gavėjas (*angl. receiver*)

  Aktorius kuris gauna pranešimą iš Aktoriaus siuntėjo. Šią abstrakcija Aktorius įgauna tik komunikacijos metu.

- **Pranešimo siuntimas** (*angl. Message passing*)

  Komunikavimo būdas tarp Aktorių, Aktorių modelio kontekste visada suprantamas kaip asinchroninis. Norint siųsti pranešimą, Aktorius siuntėjas turi žinoti Aktoriaus gavėjo adresą. Adresas išreiškiamas kaip kelias failų sistemoje, įskaitant joje esančią Aktorių sistemą iki konkrečios Aktoriaus esybės.[HMS12] Pavyzdžiui AKKA Aktorių sistemoje sukurtas Aktorius įgyja vardą tiesioginiai pagal kelią hierarchinėje medžio struktūroje.

Vardas, o tuo pačiu ir pilnas kelias, kuriuo Aktorius gali būti pasiekiamas primena įprastą kelią failų sistemoje. Dėl šios priežasties Aktoriai gali bendrauti net ir esantys skirtingose mašinose (7 pav.).



7 pav. Aktoriai skirtingose Java virtualiose mašimose (JVM).[BLO13]

- **Pranešimo apdorojimas** (*angl. message processing*)

  Po sukūrimo (būsena-*naujas*) Aktorius gali atnaujinti savo vidinę būseną į *pradėjęs darbą* tik naudodamas duomenis, gautus jam atsiųstu pranešimu. Po šio tarpinio rezultato jis gali pats siųsti pranešimus, bei apdoroti tuos, kuriuos gavo. Apdorojant pranešimus Aktoriui taikomas apribojimas - vienu metu jis gali apdoroti daugiausiai vieną pranešimą, tačiau nėra jokių apribojimų, kuriais remiantis Aktorius turėtų nuspręsti, kurią iš gautų žinučių jis turėtų apdoroti pirmiau. Apdorodamas pranešimus Aktorius gali keisti savo būseną ir elgesį, kas įtakoja veiksmus ir taisykles, kaip bus apdorojamas kitas pranešimas.

- **Pranešimų nukreipėjas** (*angl. router)*

  Tai Aktorius, kuris yra sukuriamas tam, kad užtikrintų efektyvų pranešimų tarp tam tikros srities Aktorių nukreipimo būdą. Paprastai Aktorių grupė yra sukuriama vienos paskirties uždaviniui spręsti lygiagrečiai, principas panašus kaip daugelio gijų aplinkoje keletui gijų lygiagrečiai sprendžiant vieno tipo uždavinį, tačiau tarp Aktoriaus ir gijos lygybės dėti negalima, tai skirtingi primityvai. Pranešimų nukreipėjas užtikrina efektyvų pranešimų paskirstymą jo prižiūrimiems Aktoriams. Šis Aktorius, atsižvelgdamas į pranešimų srautą, išlygiagretina savo kuruojamų pranešimų nukreipimo procesą. Pranešimų nukreipėjas nukreipia pranešimą vienam iš savo prižiūrimų Aktorių arba jų grupei, taip pat grąžina atsakymą (taip pat pranešimą) pirminio pranešimo siuntėjui.

- **Dispečeris** (*angl. dispatcher*)

  Elementas, skirtas optimizuoti sistemos resursų panaudojimą, išreiškiamas per mechanizmą kontroliuojantį Aktoriaus ar jo grupės pranešimų apdorojimą. Techninės realizacijos lygyje sutinkamas kaip turintis konfigūracijos galimybes, kurių dėka valdomi Aktoriui ar jų grupei skirti sistemos resursai (pavyzdžiui gijos).

- **Pranešimas** (*angl. message*)

  Komunikacijos tarp Aktorių elementas, skirtas informacijai perduoti. Pranešimas yra nekintantis (*angl. immutable*), atitikmeniu objektinio programavimo kalbose galėtume laikyti konstanta, tačiau tai būtų ne visai tikslu, nes prie konstantos tipo „kintamųjų“ gali prieiti įvairūs objektai, tuo tarpu pranešimą pasiekti gali tik tas Aktorius, kuriam pranešimas yra skirtas.

- **Pašto dėžutė** (*angl. mailbox*)

  Vienam Aktoriui ar Aktorių grupei skirtų pranešimų talpykla, kartais literatūroje vadinama konteineriu.

- **Aktorių krūva** (*angl. actor pool*)

  Aktorių rinkinys, apdorojantis to paties tipo pranešimus. Paprastai, tai Aktoriai, sprendžiantys to paties tipo uždavinį.

2.4. Programavimo kalbos paremtos Aktorių modeliu

Aktorių modelio programavimo kalbas vienija šie principai: objektas savyje turi ir kontroliuoja savo gijas, objektai bendrauja asinchroniškai siųsdami žinutes, žinutės dedamos į eilę ir apdorojamos savo pačių gijomis.

Ankstyvųjų programavimo kalbų, kurios iš esmės buvo paremtos Aktorių modeliu erą galima būtų pradėti skaičiuoti nuo 1975 metų, kai Carl Hewitt pristatė pirmąją Aktorių modeliui skirtą programavimo kalbą PLASMA, vėliau, kaip jos tęsinys atsirado Act 1, Act 2, Act 3 ir kitos kalbos[PAR15]. Šiame darbe neturime tikslo pateikti baigtinio Aktorių modeliu paremtų kalbų sąrašo,tačiau siekiame parodyti galimus įrankius skirtus modelio įgyvendinimui.

- PLASMA
- Act 1, Act 2, Act 3
- PLASMA II
- ALOG
- Hybrid
- SMART
- ACT++
- STUDIO
- Acttalk
- Ani
- Cantor
- Rosette

Žemiau pateikiame naujesnės Aktorių modeliui skirtos programavimo kalbos [PAR15], Iš jų būtų galima išskirti Erlang, kuri laikoma klasikine Aktorių modelio programavimo kalba. Nepaisant to, kad ji buvo pristatyta 1986, šiuo metu vis dar yra naudojama lygiagretaus programavimo sprendimams. Erlang programavimo kalba buvo viena pirmųjų įgyvendinusių Aktorių modelį ir kalbant apie modelio sprendimus iki šių dienų dažnai cituojama kaip pavyzdys. Šiuolaikinei ir vienai populiariausių Aktorių modeliu paremtai programavimo kalbai Scala, Erlang turėjo didelės įtakos.

- ABCL
- AmbientTalk
- Axum
- CAL Actor Language
- D
- E
- Elixir
- Erlang
- Fantom
- Humus
- Io
- Ptolemy Project
- Rebeca Modeling Language
- Reia
- Rust
- SALSA
- Scala
- Scratch

Vienas iš Erlang programavimo kalbos kūrėjų Joe Armstrong savo darbe „Concurrency Oriented Programming in Erlang“ teigia, kad objektinio programavimo kalbos yra skirtos objektams. Lygiagretaus programavimo kalbos skirtos lygiagretaus programavimo sprendimams, taip pabrėždamas pastarųjų kalbų išskirtinumą ir pritaikomumą (angl. „An object oriented language is a language with good support for objects. A concurrency oriented language has good support for concurrency“). Aktorių modelis savo yra ruožtu vienas iš lygiagretaus programavimo realizacijos būdų.

2.5. Karkasai ir bibliotekos skirtos Aktorių modelio įgyvendinimui

Žemiau, *1 Lentelėje* pateiktos bibliotekos ir programavimo karkasai, leidžiantys kurti Aktorių modeliu paremtus sprendimus ne tik tomis programavimo kalbomis, kurios prigimtinai pritaikytos modeliui, bet ir kitomis populiariomis programavimo kalboms, taip suteikiant kūrėjui galimybę naudoti modelį jo įprasta programavimo kalba, neskiriant laiko papildomos kalbos įsisavinimui.

Šiame darbe neturime tikslo pateikti baigtinio bibliotekų ir programavimo karkasų sąrašo, tačiau surinkome ir susisteminome informaciją apie plačiau paplitusioms programavimo kalboms skirtus įrankius Aktorių modeliui išreikšti.[N11] [KSA09] [PAR15]

1 Lentelė. Karkasai ir bibliotekos skirtos Aktorių modelio įgyvendinimui

Java	**C/C++**	**C#**	**Phython**	**Haskel**	**Ruby**	**F#**	**Scala**	**Objective-C**
▪ Akka ▪ CloudI ▪ Korus ▪ Kilim ▪ ActorFoundry ▪ Ateji PX ▪ JActor ▪ Jetlang ▪ Quasar ▪ S4 ▪ Actor Architecture ▪ Actors Guild ▪ JavAct ▪ AJ ▪ Jsasb	▪ QP ▪ CloudI ▪ SObjectizer ▪ Libprocess ▪ Actor-CPP ▪ Theron ▪ Act++ ▪ Broadway ▪ Thal ▪ Microsoft's Asynchronous Agents Library	▪ Akka.NET ▪ Actor Framework ▪ Remact.Net ▪ Retlang ▪ NAct	▪ PARLEY ▪ CloudI ▪ Pulsar ▪ Pykka ▪ Stack ▪ Less Python	▪ Haskell-Actor ▪ Cloud Haskell	▪ Celluloid ▪ CloudI ▪ Stage	▪ Akka.NET ▪ F# MailboxProcessor	▪ Akka	▪ ActorKit

3. KOMUNIKAVIMO BŪDAI

3.1. Sinchroninis ir asinchroninis komunikavimas

Aktorių modelio viena iš išskirtinių savybių yra ta, kad Aktoriai tarpusavyje komunikuoja asinchroniniu būdu. Žinutės siuntėjas visiškai nėra priklausomas nuo jos gavėjo ir atvirkščiai, todėl vienas iš jų atlikęs savo darbo dalį visai neprivalo laukti, kol gaus atsakymą, o gali toliau atlikinėti kitas užduotis, žinant, kad atsakymas bus gautas kažkada vėliau. Bene paprasčiausias asinchroninio komunikavimo pavyzdys galėtų būti elektroninis paštas. Kad vyktų komunikacija elektroniniu paštu, tarp siuntėjo ir gavėjo nėra jokios būtinybės, kad jie abu būtų prisijungę prie tinklo. Atsiųstą žinutę gavėjas gali perskaityti nebūtinai iš karto, bet žinutę jis pamatys, kai tik patikrins savo elektroninio pašto dėžutę. Viskas labai panašiai Aktorių modelio atveju, tik į mano pateiktą gyvenimišką atvejį reiktų įvesti taisyklę, kad visgi į visus elektroninius laiškus visada turi būti atsakyta, tačiau nebūtinai tokiu pat eiliškumu, kokiu laiškai buvo gauti.

Priešingybė asinchroniniam komunikavimo būdui būtų sinchroninis komunikavimas. Pavyzdžiu iš gyvenimo galėtų būti paprasčiausias pokalbis telefonu. Šiuo atveju abu komunikuojantys objektai turi turėti ryšį vienas su kitu. Tiesioginiam ryšiui nutrūkus žinutės, šiuo atveju balso pranešimo pristatymas, tampa nebeįmanomos. Be to, jei vienas iš pašnekovo padeda ragelį kol kitas dar kalba, kalbantysis gali ir nesužinoti, kad paskutinė jo perduodama informacija adresato nepasiekė.

3.1.1. Asinchronininio komunikavimo modeliai

Marco Brambilla, Giuseppe Guglielmetti, ir Christina Tziviskou straipsnyje “Asynchronous Web Services Communication Patterns in Business Protocols” be kitų komunikavimo modeliu išskiria ir šiuos tris asinchroninio komunikavimo modelius [BGT05], kurie galėtų būti įgyvendinti Aktorių modeliu.

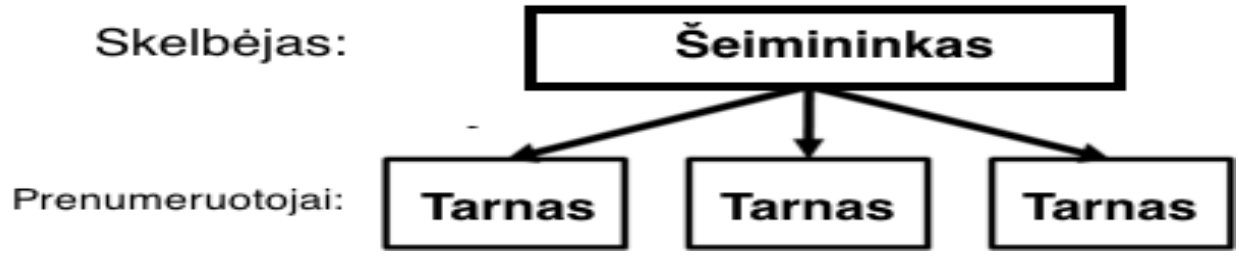
3.1.1.1. Atgalinis ryšio modelis (*angl. Callback*)

Žinutės siuntėjas, siųsdamas pranešimą gavėjui, pačioje žinutėje nurodo, kad tikisi atsakymo, ir kad jo laukia. Kai žinutės gavėjas yra pasiruošęs duoti atsakymą, jį išsiunčia siuntėjui. Gavęs atsakymą, iš žinutės gavėjo, siuntėjas yra tikras, kad jo siųstą žinutę gavėjęs

gavo. Šis metodas gali būti taikomas ir sinchroninio komunikavimo atveju, jei komunikuojantys objektai yra pasiekiami esamu momentu (pavyzdžiui abu prisijungę prie interneto tinklo),tačiau plačiai naudojamas asinchroniniam komunikavimui užtikrinti.

3.1.1.2. Skelbėjo-Prenumeruotojo modelis (*angl. Publish/Subscribe*)

Modelis įgalina siųsti grupinius pranešimus. Pranešimų gavėjai išreikštinai nusako pageidavimą gauti pranešimus. Paprastai konfigūruojama kokio tipo pranešimus pageidaujama gauti, visus ar filtruojant pagal tam tikras kategorijas. Skelbėjas, turėdamas informacijos, prenumeratoriams visiems iš karto išsiunčia pranešimą,. Taigi, tą patį pranešimą gauna visi prenumeratoriai, kurie buvo užsiprenumeravę gauti tokios kategorijos žinutes. Modelis gali būti įgyvendinamas panaudojant Šeimininko-tarno išlygiagretinimo schemą, kuri pristatyta toliau šiame darbe.



8 pav. Skelbėjo prenumeratoriaus modelis išreikštas Šeimininko - tarno išlygiagretinimo modeliu.

3.1.1.3. Apklausos modelis (*angl. Polling*)

Žinutės gavėjas nusiunčia žinutės siuntėjui prašymą gauti žinutę. Yra sutariama, kad gauta iš siuntėjo žinutė bus patalpinama erdvėje, prie kurios gali prieiti ir žinutės gavėjas. Žinutės gavėjas periodiškai tikrina, ar nėra atsiųstų naujų žinių, kurias reikia apdoroti.

4. PROGRAMŲ IŠLYGIAGRETINIMAS

Išlygiagretinimas - tai būsena, kai programos vykdymo vienetas vienu metu atlieka daugiau negu vieną užduotį - tą skaičių užduočių vykdo lygiagrečiai, taip pasiekdamas proceso greitinimą, kitais žodžiais tariant, dėka lygiagretaus veikimo, atlieka daugiau darbo per tokį pat arba mažesnį laiko vienetą. Išlygiagretinimas gali būti pritaikomas tiek aparatūrinėje aplinkoje, tiek programų sistemos lygmeny.

Būtų galima išskirti šiuos išlygiagretinimas tipus:

- *Bitų lygio išlygiagretinimas* (*angl. bit level parallelism*). Taikymo sritis: mikroprocesoriai;
- *Instrukcijų lygio išlygiagretinimas* (*angl. instruction level parallelism*). Taikymo sritis: kompiliatoriai;
- *Operacinės sistemos lygio išlygiagretinimas* (*angl. multiprocessing, multi-tasking level parallelism*). Taikymo sritis: operacinės sistemos;
- *Serverio lygio išlygiagretinimas* (*angl. high performens computing, clustering*). Taikymo sritis: serveriai;
- *Gijų lygio išlygiagretinimas* (*angl. multi-threading*). Taikymo sritis: programinio kodo vykdymas;
- *Duomenų lygio išlygiagretinimas* (*angl. data parallelism).* Taikymo sritis: išskirstytos duomenų bazės;
- *Užduoties lygio išlygiagretinimas* (*angl. task parallelism*). Taikymo sritis: programinio kodo (neretai išskirstytų aplikacijų) vykdymas.

Šio magistrinio darbo kontekste, nagrinėjant Aktorių modelio išlygiagretinimo sprendimus, analizuosime tik Gijų ir Užduoties lygių išlygiagretimo schemas.

4.1. Faktoriai, įtakojantys lygiagrečios programos veikimą

Kaip minėjome anksčiau, lygiagretus programavimas yra aibė veiksmų, kurie visi kartu ir kiekvienas atskirai prisideda prie lygiagrečiai vykdomų procesų, siekiant paspartinti programos atlikimą. C.M. Pancake [P96] išskiria tris pagrindinius komponentus turinčius įtakos išlygiagretintos programos veikimui:

- *techninės įrangos platforma;*
- *programavimo kalba;*
- *problemos, arba kitaip tariant uždavinio sprendimo būdas.*

Techninės įrangos ir programavimo kalbų aspektai, turintys įtakos programos veikimui, šiame magistriniame darbe plačiau nebus nagrinėjami, tačiau detaliau apžvelgsime problemos sprendimo būdus, kas yra vienas iš kertinių faktorių įtakojančių išlygiagretintos programos efektyvų vykdymą.

- ***Problemos sprendimo būdas*** yra suprantamas kaip tinkamiausio algoritmo (-ų), bei teisingo duomenų rinkinio pasirinkimas, norint pasiekti efektyviausią išlygiagretinimo rezultatą konkrečiam uždaviniui spręsti. Jorge Luis Ortega Arjona savo disertacijoje [OA06] pažymi, kad priklausomai nuo to, kokie problemos sprendimo būdai ir metodai pasirenkami programos išlygiagretinimui, labai stipriai priklauso šios programos vykdymo sėkmė arba nesėkmė. Taip pat autorius pastebi, kad tuo atveju, kai išlygiagretintos programos spartesnio veikimo pagrindas yra algoritmų, bei duomenų dekomponavimas, tuomet lygiagrečios programos veikimas stipriai įtakojamas tuo, kokiu eiliškumu yra vykdomos užduoties algoritmo sudedamosios dalys, nepriklausomai nuo to, kokio tipo problemą bandoma išspręsti.

4.2. Lygiagretaus programavimo modeliai paremti duomenų ir operacijų dekomponavimu

Siekiant apibrėžti struktūrą, kaip turėtų būti konstruojama lygiagreti programa, daugelis autorių išskiria tris pagrindinius modelius:

- *Funkcinis išlygiagretinimą;*
- *Duomenų išlygiagretinimą;*
- *Veiklos išlygiagretinimą*.

Pastarasis yra glaudžiai susijęs tiek su funkciniu, tiek su duomenų išlygiagretinimu [O06]. Dekompozicija yra visų trijų modelių pagrindas. Pažymėtina, kad visi trys jie glaudžiai susiję ir sudaro vieną bendrą grandinę, formuojant lygiagrečios programos sprendimus.

4.2.1. Funkcinis išlygiagretinimas

Funkcinio išlygiagretinimo modelis (*angl. Functional parallelism),* dar literatūroje sutinkamas kaip užduoties išlygiagretinimas (*angl. task parallelism*), skirtas algoritmų dekompozicijai. Modelio tikslas ir idėja yra išskaidyti užduoties sprendimo algoritmą į atskiras, viena nuo kitos kuo labiau nepriklausomas dalis, kurios gali būti vykdomos visos tuo pačiu metu.

Kai nepriklausomos dalys yra išskirtos, tikrinama ar nepažeisti duomenų reikalavimai kiekvienai daliai atskirai. Jei duomenų reikalavimai kiekvienai atskirtų dalių taip pat yra tinkamai dekomponuoti, tuomet užduoties dekompozicija gali būti laikoma baigta.

Funkcinio išlygiagretinimo modelio kontekste visos išskaidytos nepriklausomos algoritmo dalys darbą pradeda vienu metu, tačiau kartais pradžioje daliai jų gali tekti laukti, kol duomenys bus paruošti vykdymui, taip yra todėl, kad iš pradžių gali būti poreikis trūkstamus duomenis paruošti (apskaičiuoti, surinkti ir pan.). [CG88] [P96]

4.2.2. Duomenų išlygiagretinimas

Duomenų išlygiagretinimo modelis (*angl. data parallelism)* - paremtas duomenų dekompozicijos idėja. Šias išskaidytas duomenų dalis naudoja ir/arba kuria *funkciniame išlygiagretinimo modelyje* aptartos algoritmų dalys. Rekomenduojama duomenis išskaidyti į smulkesnes, geriausia kuo artimesnes vienodam dydžiui dalis. Duomenys išskaidomi, pagal tai, kad išskaidytos algoritmų dalys galėtų veikti nepriklausomai ir kuo efektyviau. Kartais tam tikros operacijos gali reikalauti duomenų iš skirtingų algoritmo elementų (išskaidytų dalių), todėl atsiranda poreikis komunikacijai tarp šių išskaidytų užduoties vykdymo komponentų. Literatūroje šį modelį taip pat galima sutikti *rezultato išlygiagretinimo* vardu (*angl. result parallelism*).

4.2.3. Vykdymo išlygiagretinimas

Vykdymo išlygiagretinimo modelis (*angl. activity parallelism*), taip pat žinomas kaip duomenų srauto (*angl. data flow parallelism*) arba darbotvarkės išlygiagretinimas (*angl. agenda parallelism*) reikalauja tiek duomenų, tiek algoritmų dekompozicijos. Priklausomai nuo dalykinės srities logikos, tam tikros išskaidyto algoritmo dalys dirba su tam tikra, taip pat išskaidyta, dalimi duomenų. Skirtingos užduočių dalys dirba su skirtingomis duomenų dalimis. Apdorojus tam tikrą duomenų dalį grąžinamas rezultatas, jei dar yra neapdorotų duomenų, darbas tęsiamas. Kartais tam tikra darbų dalis gali būti pradėta tik po to, kai prieš tai jau buvo atlikta ankstesnė darbo dalis, tačiau tiek prieš tai vykdytų išskaidytų užduočių rinkinys, tiek vėlesnis yra vykdomi lygiagrečiai, kiekvienam algoritmo darbo vienetui gaunant, o vėliau, kaip rezultatą grąžinant, dekomponuotą duomenų dalį. *Vykdymo išlygiagretinimas* neretai išreiškiamas kaip Gamintojo - vartotojo modelis (*angl. Producer-consumer chain),* kurį detaliau aptarsime sekančiuose darbo skyriuose.

4.3. Išlygiagretinimo sprendimų klasifikacija

Žemiau pateikiame klasifikacijos lentelę, kurioje išvardinti išlygiagretinimo sprendimų šablonų (*angl. patterns*) rinkiniai. Tikslumo dėlei, išlygiagretinimo modeliai ir praktikos pateikiami anglų kalba, taip, kaip juos galima dažniausiai sutikti literatūroje. Suklasifikuotas rinkinys pateikiamas siekiant parodyti visuminį vaizdą gerųjų praktikų išlygiagretinimo kontekste.

2 Lentelė. Išlygiagretinimo gerųjų praktikų klasifikacija

.

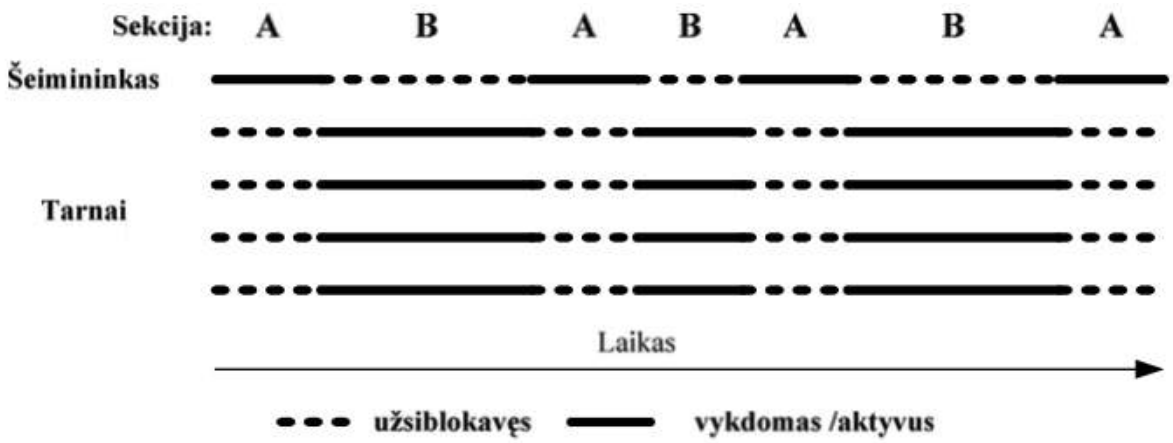
4.4. Išlygiagretinino schemos

Šiek tiek detaliau apžvelgsime keturias, bene populiariausias, lygiagretaus programavimo schemas, kurios potencialiai tinka įgyvendinti Aktorių modeliu.

- *Šeimininko – tarno (angl. master-slave).*
- *Darbų krūvos (angl. thread pool)*
- *Konvejerio (angl. pipeline)*
- *Gamintojas-Vartotojas (angl. Producer-consumer*)
- *Skaldyk ir valdyk (angl. divide-and-conquer)*

4.4.1. Šeimininko – tarno schema

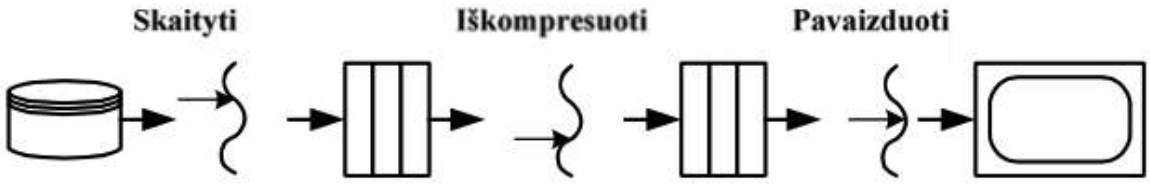
Schemos esmę sudaro toks veikimo principas, kai viena gija, kuri vadinama šeimininku, startuoja kitas gijas - vadinama tarnais. Tarnams darbai paskirstomi po lygiai. Šeimininkas, paskirstęs darbus, turi sulaukti kol visi tarnai atliks jiems pavestus darbus. Tarnams, baigus darbus, šeimininkas agreguoja gautus tarpinius uždavinio rezultatus, kuriuos pateikė kiekvienas iš tarnų. Paprastai darbas yra išskaidomas į daugiau dalių, negu yra tarnų, tokiu atveju vienas tarnas pateikia nuo 1 iki n tarpinių rezultatų. Šeimininkas gavęs rezultatus, gali arba baigti darbą, arba kartoti procesą cikle. Ši išlygiagretinimo schema tinkamiausia tokiuose uždaviniuose, kuriuose atliekamo darbo apimtis žinoma iš anksto. Pačios programos spartėjimo efektas priklauso nuo galimybės darbus išskaidyti į kuo mažiau dalių (*angl. granularity*), žinoma, reiktų atsižvelgti ir į tai, kad komunikacija tarp šeimininko ir tarnų taip pat kainuoja laiko, todėl laikas, skirtas komunikacijai, bei darbų skaidymas į kuo smulkesnes dalis turėtų būti optimalus. Klasikinio lygiagretaus programavimo atveju gijų sinchronizacijai naudojamas barjeras, tačiau Aktorių modelio atveju, tai nėra aktualu, nes šie primityvai, kaip minėjome anksčiau, komunikuoja asinchroniniu būdu.



9 pav. Šeimininko- tarno išlygiagretinimo schema. [VAI07]

4.4.2. Darbų krūvos schema

Darbas kaip ir Šeimininko-tarno atveju suskaidomas į smulkesnes dalis. Patogumo dėlei galime tokius išskaidytus darbus pavadinti užduotimis. Šios užduotys patalpinamos gijoms ar kitiems primityvams (pvz. Aktoriams) pasiekiamoje erdvėje. Visos šios užduotys laikomos darbų krūva. Gijos ar kiti primityvai ima iš eilės po vieną užduotį, ją atlikus iš eilės imama kita užduotis. Darbo procesas laikomas baigtu, kai darbų krūvoje nebelieka neužbaigtų užduočių, o visi dirbę primityvai tampa laisvi. Uždavinio sprendimo eigoje gali atsirasti naujų užduočių, kurios talpinamos į tą pačią darbų krūvą, eigoje gali atsirasti naujų darbų, kurie talpinami į tą pačią krūvą. Esminis skirtumas nuo negu *Šeimininko - tarno* schemos, yra tas, kad užduotys yra skirtingos, todėl klasikiniu lygiagretaus programavimo atveju sumažėja gijų prastovos, tačiau Aktorių modelio atveju dėl minėto asinchroninio komunikavimo būdo tai neaktualu.
Programuojant darbų krūvos realizacija – paprastai sinchronizuota eilė.



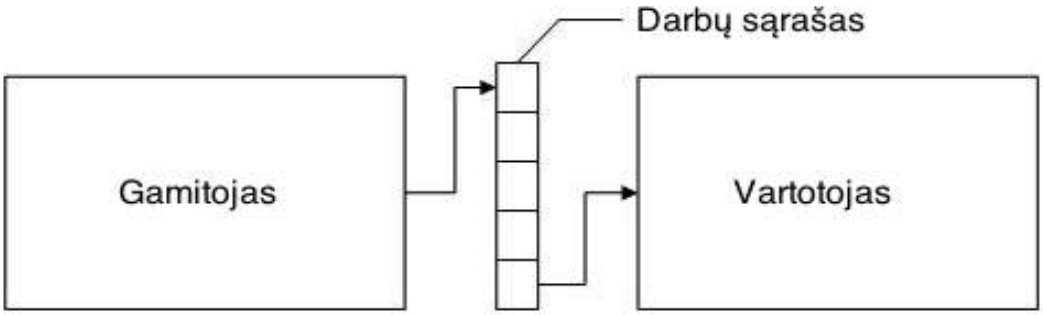
10 pav. Darbų krūvos išlygiagretinimo schema. Paveikslėlio pateiktis (*angl. rendering*) [VAI07]

4.4.3. Konvejerio schema

Šioje schemoje kiekvienas darbas, skaidomas į smulkesnes dalis – užduotis, jos savo ruožtu yra perduodamos gijų ar kitų primityvų konvejeriui. Kiekvienas primityvas atlieka tik tam tikrą užduoties dalį. Atlikta užduoties dalis perduodama konvejeriu kitam primityvui, kad šis atliktų savąją dalį. Tai tęsiasi, tol, kol uždavinys yra pilnai išsprendžiamas, o dirbę primityvai tampa laisvi. Schema tipiška klasikiniam konvejeriui naudojamam pramonėje, kuomet kiekvienas prie konvejerio dirbantis darbininkas atlikęs tam tikrą darbo dalį, perduoda pusgaminį kitam, kol galiausiai pagaminamas produktas. Daugeliu atveju atliekami darbai kiekvienoje konvejerio grandyje yra skirtingi, tačiau galimi ir homogeniniai konvejeriai.
Realizuojant šią schemą, kiekviena gretimų primityvų pora sąveikauja kaip *gamintojas-vartotojas*. [VAI07]

4.4.4. Gamintojas-Vartotojas

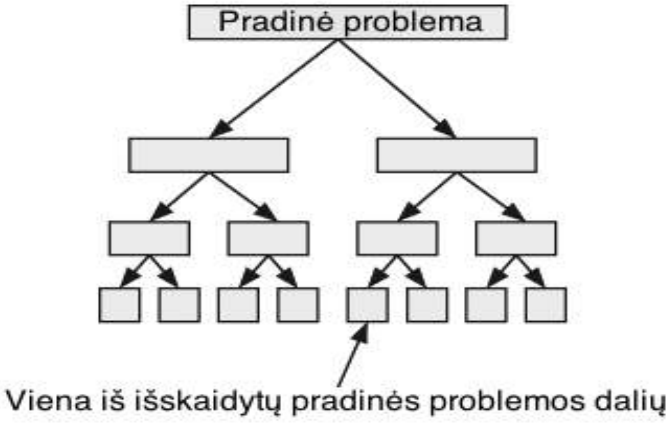
Schema gan paprasta, joje sąveikauja pora primityvų *gamintojas ir vartotojas*. Vartotojas gali vartoti tik tuomet, kai gamintojas yra kažką pagaminęs. Kitais žodžiais tariant, jei gamintojas nėra perdavęs užduočių vartotojui, šis jos paprasčiausiai neturi galimybės vykdyti. Priklausomai nuo uždavinių tipo, realizuojant schemą gali būti naudojama keletas principų, pavyzdžiui gamintojas „gamina“ tik po vieną užduotį („gaminį“), vartotojui jį atlikus, („suvartojus“) pateikia kitą, arba gamintojas „gamina“ n skaičių užduočių ir tik jį pasiekęs nustoja gaminti, tuo tarpu vartotojas atlikęs užduotį apie tai praneša gamintojui, taip šis žino, kad vėl laikas gaminti.



11 pav. Gamintojo-vartotojo išlygiagretinimo schema.

4.4.5. Skaldyk ir valdyk

Skaldyk ir valdyk– schema, paremta idėja, kad darbai būtų išskaidomi į mažesnes dalis ir kiekviena iš jų sprendžiama lygiagrečiai. Skyriuje „Lygiagretaus programavimo modeliai“ paremti duomenų ir operacijų dekomponavimu pateikėme ir detaliau aptarėme tris skirtingus modelius - funkcinį, duomenų ir vykdymo.



12 pav. Skaldyk ir valdyk išlygiagretinimo schema.

Aptartų darbo dalių apžvalga

Šioje magistro darbo dalyje apžvelgėme Aktorių modelį, esmines Aktorių modelio savybes, pateikėme pradinį Aktorių modelio notacijų rinkinį, bei asinchroninio komunikavimo būdus, kas laikoma viena iš kertinių Aktorių modelio savybių. Taip pat buvo apžvelgti lygiagretaus programavimo tipai, nagrinėti faktoriai, turintys įtakos lygiagrečiai veikiančios programos efektyvumui, išlygiagretinimo sprendimai, pateikta geruųjų išlygiagretinimo praktikų klasifikacija, pasirinktos tipinės išlygiagretinimo schemos tolesniam jų išpildymui Aktorių modelio galimybėmis. Tolesniuose magistro darbo skyriuose bus nagrinėjama kuo remiantis būtų galima identifikuoti potencialius Aktorius, kaip techninių spendimų priemonę ir kokias gerąsias praktikas taikyti, siekiant, kad Aktoriais grįsta sistema būtų kuo našesnė.

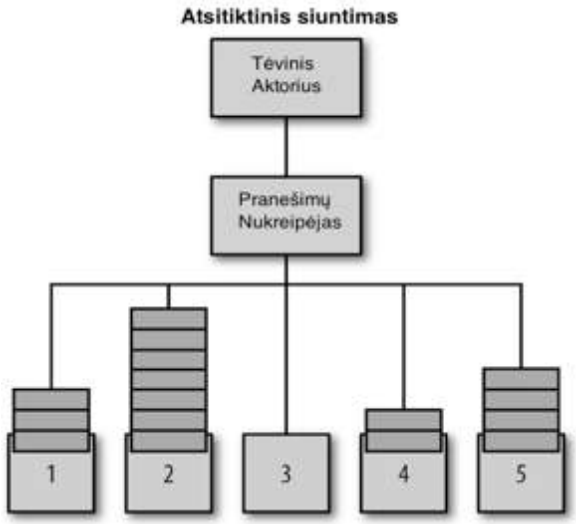
5. GEROSIOS PRAKTIKOS IR PATARIMAI AKTORIŲ MODELIO NAUDOJIMUI

5.1. Pranešimų paskirstymo strategijos

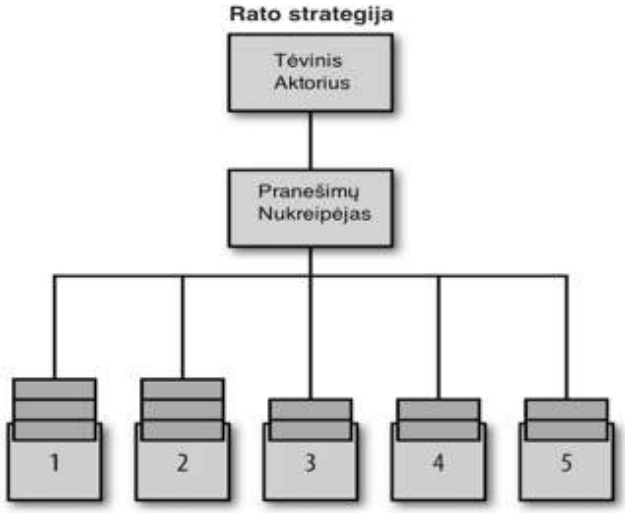
Galimos įvairios strategijos, kuriomis vadovaujantis yra skirstomi darbai, Aktorių modelio terminais kalbant, tai būtų siunčiamų žinučių paskirstymas tarp atitinkamų Aktorių egzempliorių. Žemiau pateikiame 5 strategijas, kurias galima taikyti pranešimų paskirstymui:

- *Atsitiktinio siuntimo;*
- *Rato;*
- *Mažiausiai užpildytos pašto dėžutės;*
- *Paskleidimo;*
- *Paskleidimo-surinkimo;*

Schemose pateikiame gerąsias Aktorių pranešimų paskirstymo praktikas, tėvinis aktorius siunčia pranešimą Aktoriui darbininkui, tačiau netiesiogiai, o per Aktorių - pranešimų nukreipėją, kuris naudodamasis viena iš jam apibrėžtų strategijų skirsto pranešimus tarp atskirų vaikinių Aktorių egzempliorių. Schemose pažymėti kvadratai su numeruoti nuo 1 iki 5 (arba nesunumeruoti Paskleidimo-Surinkimo strategijos atveju) reprezentuoja Aktorių egzempliorius, tuo tarpu plytelės virš jų pranešimus esančius konkretaus Aktoriaus pašto dėžutėje. Pranešimų nukreipėjas naudojamas siekiant įgyvendinti gerąsias pranešimų paskirstymo praktikas tarp Aktorių darbininkų.



13 pav. Atsitiktinis siuntimas



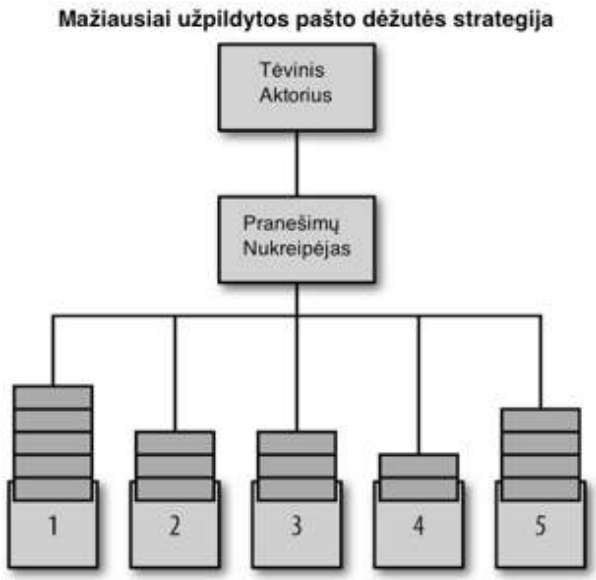
14 pav. Rato strategija

5.1.1. Atsitiktinis siuntimas

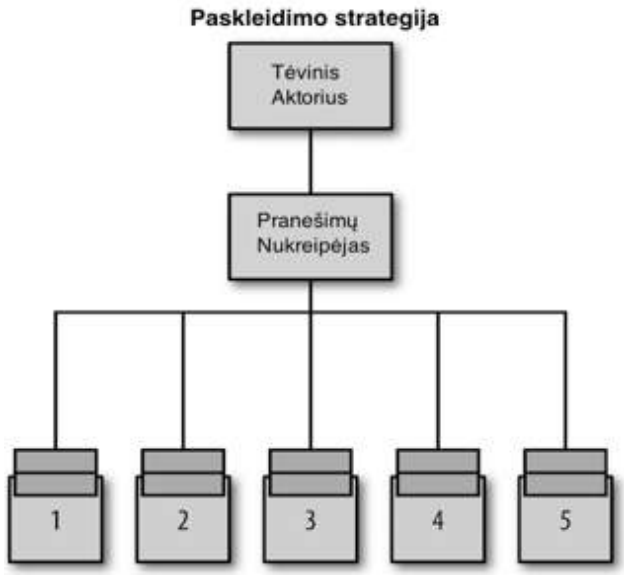
Atsitiktinio siuntimo (*angl. random*) strategija leidžia nesusidaryti taip vadinam butelio kakliuko efektui (*ang. bottleneck*), nes prieš siunčiant pranešimus į Aktorių pašto dėžutes nereikia atlikti jokių papildomų patikrų, kas kainuoja laiko ir žinoma resursų. Kita vertus, ši strategija netinkama, tuo atveju jei siekiant didesnio skaliuojamumo ar turint kitų tikslų sukuriamas didesnis Aktorių skaičius, tikintis, kad darbai bus atlikti greičiau. Atsitiktinis žinučių paskirstymas gali sąlygoti situaciją, kai dalis egzempliorių bus užkrauti darbu, tuo tarpu, dalis jų darbo niekada negaus.

5.1.2.Rato strategija

Rato *(angl. Round-robin)* strategijos atveju žinutės skirstomos eilės kiekvienam Aktoriui po lygiai, tai puikiai tinka ir norima tolygiai paskirstyti apkrovas tarp Aktorių egzempliorių. Strategija labai tinkama tuo atveju, kai Aktoriams visada paskiriamas atlikti tas pats darbas, taip pat kai darbų atlikimo greitis gali tiesiogiai priklausyti nuo procesoriaus pajėgumo.



15 pav. Mažiausiai užpildytos dėžutės strategija



16 pav. Paskleidimo strategija

5.1.3. Mažiausiai užpildyta dėžutė

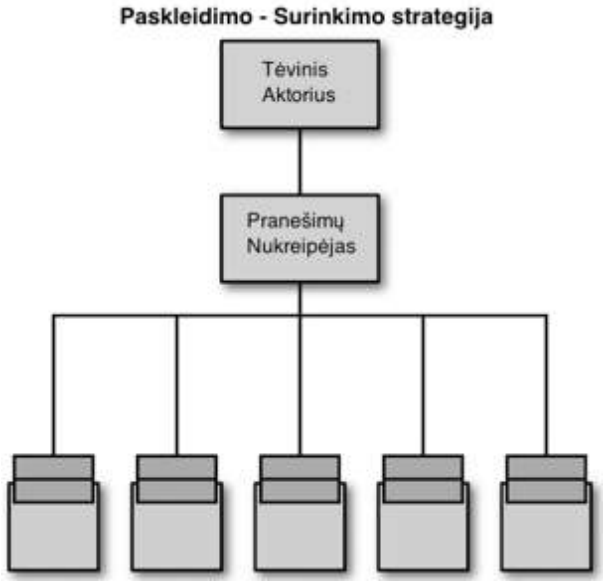
Mažiausiai užpildytos pašto dėžutės strategijos esmė, kad pranešimą nukreipiame tam Aktoriaus egzemplioriui, kurio pašto dėžutė yra mažiausiai užpildyta, darant išvadą, kad Aktoriai turintys labiau užpildytas pašto dėžutes yra labiau apkrauti darbu ir atvirkščiai. Nors iš pirmo žvilgsnio gali pasirodyti, kad toks sprendimas tiktų visiems atvejams, deja taip nėra. Mažesnis pranešimų skaičius pašto dėžutėje ne visada reiškia, kad Aktorius greičiau apdoros siunčiamą pranešimą, lyginant su kitais Aktoriais, kurių pašto dėžutės labiau užpildytos. Gali būti, kad santykinai mažiau užpildytos pašto dėžutės savininkas atlieka darbus, kurie yra sudėtingesni ir reikalauja daugiau laiko, taigi gali būti kad Aktorius turintis, kurio pašto dėžutė beveik užpildyta einamąjį pranešimą atliks greičiau.

Ši strategija panašiai, kaip Round Robin atveju tinka kai Aktoriai turi atlikti visiškai tuos pačius darbus, tačiau darbai natūraliai pasižymi blokavimusi[ALL13]. Ši strategija gali būti mažiau tinkama arba visai netinkama tais atvejais kai Aktoriai išsidėstę skirtose mašinose, nes pranešimų paskirstytojas tiesiog gali nežinoti nutolusio (*angl. distributed*) Aktoriaus pašto dėžutės užimtumo, arba tam sužinoti užtrankantis laikas yra neadekvačiai didelis. To priežastis paprasta, Aktorius esantis kitoje mašinoje, gali būti pasiekiamas tinklo pagalba, tačiau komunikacijos laikas tikėtina bus nepriimtinas.

5.1.4. Paskleidimo strategija

Paskleidimo (*angl. broadcast*) strategija įgyvendinama tokiu būdu, kai pranešimas yra išsiunčiamas visiems Aktoriams, kuriuos reguliuoja Aktorius - Pranešimų nukreipėjas. Strategija tinkama tais atvejais, kai reikia paskirstyti darbus tarp skirtingose mašinose esančių Aktorių, kurie atsakingi už skirtingus darbus arba skirtingas darbų dalis.

Iš pirmo žvilgsnio gali pasirodyti, kad šiuo atveju visų Aktorių pašto dėžutės visada turės vienodą kiekį žinučių, tačiau dėl skirtingų pranešimų apdorojimo tempų, žinučių kiekiai esamu momentu skirsis. Taip pat maksimalus priimamų žinučių kiekis gali būti valdomas, apsirašant atitinkamas taisykles pašto dėžutėms. Reiktų suprasti ir tai, kad išsiųstą pranešimą Aktoriai esantys skirtingose mašinose gaus skirtingu laiko momentu, tam įtakos turės tiek tinklo pralaidumo savybės, tiek fizinis atstumas tarp skirtingų mašinų.



17 pav. Paskleidimo-surinkimo strategija

5.1.5. Paskleidimo-surinkimo strategija

Paskleidimo-surinkimo strategija (*angl. Scatter-GatherFirstCompletedOf*), naudojama, kai pranešimas yra išsiunčiamas visiems Pranešimo nukreipėjo prižiūrimiems Aktoriams, tačiau vėliau apdorojamas tik pirmas atgal gautas atsakymas. Strategija tinkama, kai norima gauti greita atsakymą, todėl pranešimas išsiunčiamas daugeliui Aktorių netikrinant, kuris iš jų potencialiai galėtų atlikti atlikti užduotį greičiausiai, gavus pirmą atsakymą, atsakymai iš likusių Aktorių ignoruojami.

Skirstant darbus tarp Aktorių siunčiami pranešimai paprastai yra komandos, į kurias Aktoriai moka reaguoti atliekant konkrečia užduotį ir jei to reikalaujama grąžinant atsakymą siuntėjui. Pranešimas savyje turi turėti duomenis, kuriais vadovaudamasis Aktorius atliks atitinkamą užduotį. Užduotys turi būti suprojektuotos taip, kad nepriklausomai nuo to, kuris Aktoriaus egzempliorius ją atliks, bei kiek kartų jos bus atliekamas, paduodant tuos pačius parametrus rezultatas turi būti gaunamas toks, be jokio šalutinio efekto. (*angl. idempotent tasks*)

6. AKTORIŲ IDENTIFIKAVIMAS

Kuriant sistemą, paremtą Aktorių modeliu neretai iškyla klausimas, nuo ko pradėti? Tiek mokslinėje literatūroje, tiek kituose šaltiniuose galima rasti atskirai aptariamus Aktorių modelio teikiamus privalumus, bei trūkumus, būdus ir priemones, kuriomis, galima realizuoti modelio savybes. Taip pat galima rasti atskirų patarimų, kaip elgtis vienoje ar kitoje situacijoje, tačiau medžiagos, kuri detaliai nurodytų, kaip turėtų elgtis sistemų kūrėjas, turėdamas konkrečius panaudojimo atvejus, kaip nuo egzistuojančios problemos pereiti prie Aktoriais paremtos programų sistemos, kurios iš šios sistemos dalių turėtų būti įgyvendintos Aktoriais, kaip žinoti, kiek Aktorių kurti ir kokiais kriterijais vadovaujantis juos identifikuoti kaip sistemos realizacinius elementus. Kokios programų sistemų kūrimo gerosios praktikos galėtų būti pritaikytos įgyvendinant Aktorių modelio sprendimus. Tokie ir panašūs klausimai kyla tiek programuotojui, tiek sistemos architektui, ketinant kuriamai sistemai pritaikyti Aktorių modelio savybes.

Kadangi struktūrizuotų atsakymų į šiuos klausimus rasti nepavyko, nusprendėme išanalizuoti šiuo metu pasiekiamą informaciją mokslinėje literatūroje, susieti ją su praktikų pateikiamomis įžvalgomis ir šiame darbe pateikti pradinį informacijos rinkinį, kuris leistų susivokti nuo ko pradėti, kokios žingsnius reikia atlikti, kokias gerąsias praktikas panaudoti, atsižvelgiant tiek į praktikų jau išmoktas pamokas, tiek į mokslininkų pateikiamas įžvalgas.

Vieni pirmųjų klausimų pradedant galvoti apie sistemos projektavimą kyla kaip ir į kokias dalis išskaidyti problemą, kad išskaidytos dalys būtų tinkamos Aktorių modelio įgyvendinimui, kas konkrečiu panaudojimo atveju bus Aktorius, kokie faktoriai įtakoja Aktorių kiekį, kokius ryšius turės šie Aktoriai ir kokia jų tarpusavio hierarchija turėtų egzistuoti.

Šioje magistro darbo dalyje ieškosime būdų kaip identifikuoti Aktorius, sistemos reikalavimų įgyvendinimui.

6.1. Aktorius nėra gija

Ieškant atspirties taško, viena pirmųjų galinčių kilti minčių, galbūt Aktorius galėtume tapatinti su gija, vykdančia konkrečias užduotis. Gija veikia konkretaus proceso ribose ir nėra žinoma už jo ribų, tuo tarpu Aktorius gali būti pasiekiamas netgi jam esant kitoje Java virtualioje mašinoje, taigi tik pradėję svarstyti suprantame, kad gija ir Aktorius negali būti vienas ir tas pats.

Pažvelgę giliau kokiais principais veikia Aktorius pastebėsime, kad Aktorius turi savo giją, kurią valdo, tačiau ją turi ne visada. Jei sukurtume 1000 Aktorių, tai nereiškia, kad 1000 užsiblokavusių gijų lauks kada joms bus darbo, ir tik jam atsiradus pradės jį atlikinėti. Aktorius gauna giją tik tada kai darbo yra, ir nors konkretaus darbo vykdymui gija kaip resursas bus naudojama, tačiau Aktoriaus valdoma gija turi visai kitokią sampratą lyginant su klasikiniuose lygiagretaus programavimo sprendimuose sutinkamomis gijomis, kurios dėl sinchroninio komunikavimo principų neišvengia blokavimosi. Jei Aktorius neturi darbo, pvz.: eilėje nėra jokių žinučių, kurias reiktų apdoroti, šio Aktoriaus gija yra panaudojama kitų Aktorių darbams atlikti, todėl turimi resursai išnaudojami kur kas efektyviau. Aktorius gali apdoroti tik vieną žinutę vienu metu, apdorojamą žinutę vienu metu pasiekti gali tik viena gija. Taip pat reiktų pastebėti, kad jei Aktorius turėtų apdoroti tris žinutes A, B ir C, tai nereiškia, kad visos jos bus apdorotos vienos ir tos pačios gijos, iš tiesų, jos gali būti apdorotos trijų skirtingų gijų, tačiau yra garantuojama, kad visos trys žinutės bus apdorotos iš eilės viena paskui kitą. Aktoriai dalinasi gijas, todėl jų visada bus mažiau nei Aktorių, išskyrus išskirtinius atvejus, kad realizacija reikalauja konkrečiam Aktoriui turėti vieną dedikuotą giją – ši situacija ženkliai sumažintų sistemos skaliuojamumą, todėl šiame darbe daugiau kalbėsime apie sprendimus paremtus gijų dalinimusi

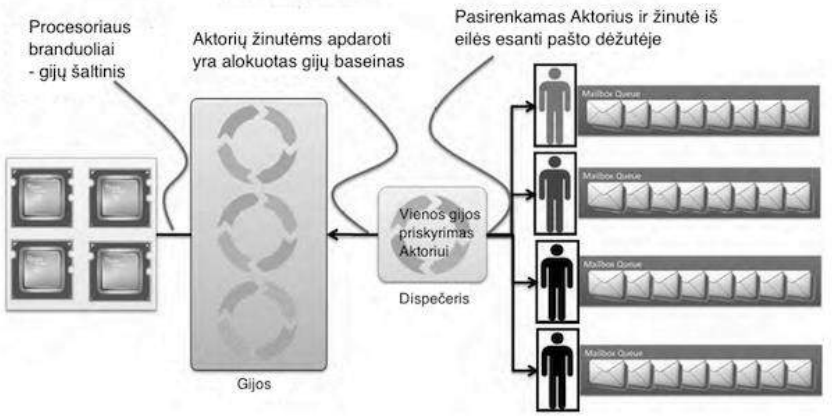
Palyginimui Aktoriaus gija veikia taip, kaip padavėjas restorane - jis aptarnauja klientą, kai jam to reikia, tačiau nesėdi ir nevalgo su juo kartu už vieno stalo. Padavėjas aptarnavęs klientą iškart eina aptarnauti kito staliuko, klientui pageidaujant dar kažko užsisakyti padavėjas vėl iškviečiamas, tačiau nebūtinai ateis tas pats, kuris aptarnavo prieš tai.

Siųsdami asinchronines žinutes Aktoriai įgyvendina „iššoviau-ir-pamiršau“ (*angl. „Fire-and-forget“*) modelį. Trumpa modelio iliustracija galėtų šis paprastas pavyzdys: daug žmonių visi vienu metu negali paskambinti telefonu vienam ir tam pačiam adresatui, tačiau jie visi gali adresatui nusiųsti SMS žinutę, Aktorių žinutės šiuo atveju panašios į pavyzdyje aptartas SMS žinutes .

Apibendrinant galime dar sykį patvirtinti, kad Aktorius ir gija neturi tiesioginės sąsajos, kitaip tariant gija neidentifikuoja konkretaus Aktoriaus, o konkreti gija gali būti tapatinama su Aktoriumi, tik tuo metu, kai gijos resursas yra naudojamas Aktoriaus užduoties vykdymui.

6.2. Kompiuterio resursai: gijos ir Aktoriai

Kad aplikacija galėtų greitai aptarnauti didelį kiekį vartotojų turi būti pakankamas resursų kiekis. Kai kalbame apie kompiuterio resursus visų pirma turime galvoje procesoriaus ir atminties resursus, nuo jų priklauso tiek galimas gijų kiekis, tiek maksimalus Aktorių skaičius.



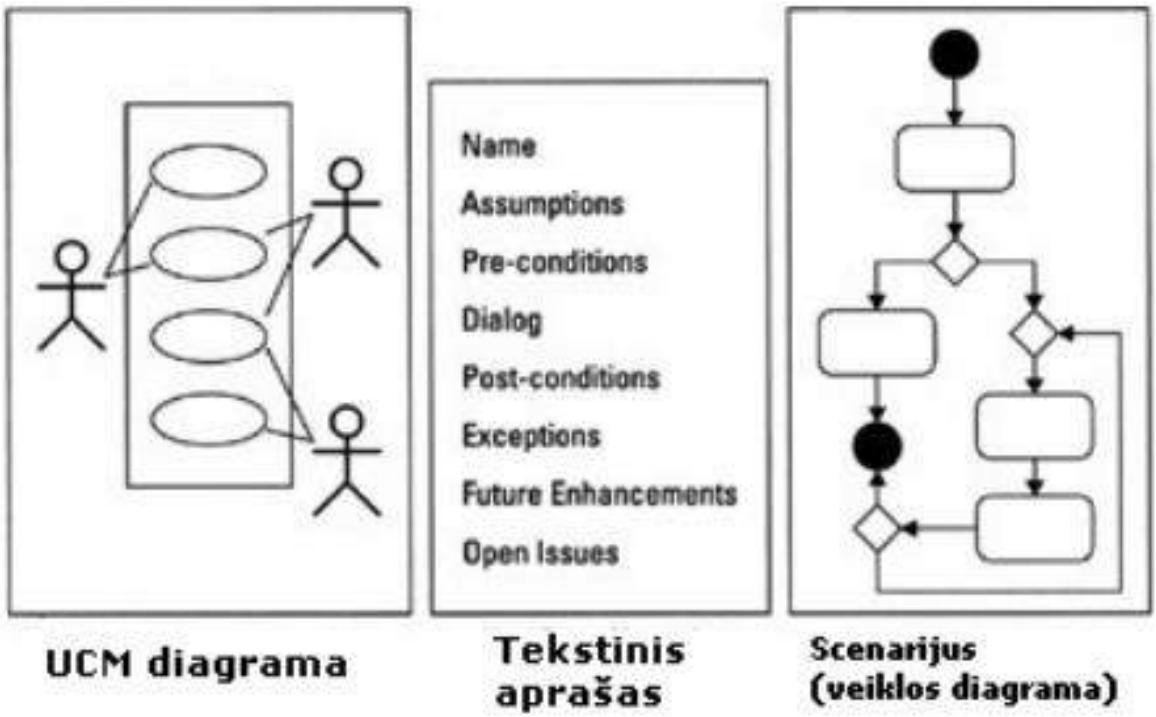
18 pav. Aktorius ir gija [M015]

Įsivaizduokime dar vieną situaciją 100 000 globalaus banko vartotojų vienu bando atlikti pinigų pavedimą pasinaudodami internetinės bankininkystės paslauga. Kad visi vienu metu galėtų tai padaryti, jiems reiktų paskirti bent 100 000 gijų. Tokiam kiekiui atskirų gijų reiks pakankamai procesorių, atminties resursų. Siekiant sumažinti reikalaujamų resursų kiekį, galima panaudoti Aktorius, kurie gijas naudoja labai efektyviai, todėl resursų šiuo atveju reiktų mažiau, kad pasiekti tą patį rezultatą. Jei reikalaujamas resursų kiekis yra didelis turimam kiekiui, pavyzdžiui turimi pajėgumai gali suteikti tik 20 000 gijų vienu metu, vartotojui, kurie nepatenka į pirmąjį 20 000 gijų sąrašą, turės laukti eilėje, kol pirmosios gijos atliks savo darbą ir galės aptarnauti naujus vartotoją. Tuo tarpu Aktorių modelio atveju, dėl efektyvaus resursų išnaudojimo sistemos veikimas gali būti kur kas spartesnis. Kaip pavyzdį galime pateikti, kad sistema, kuri pajėgi palaikyti iki 5 000 tuo pat metu aktyvių gijų – yra pajėgi palaikyti net iki 1 200 000 tuo pat metu aktyvių Aktorių. [HO07]

6.3. Nuo panaudojimo atvejo prie Aktorių

Pradedant galvoti į kokias dalis turėtų būti skaidoma problema, neretai kyla noras problemą dekomponuoti remiantis objektinio programavimo žiniomis, tačiau kiek šie principai gali būti naudingi Aktorių modelio kontekste? Pamėginkime pasinaudoti objektinio projektavimo principais, šie principai žinomi tiems, kam tekę susidurti su objektinio programavimo paradigma, be to jie gali būti gan intuityviai suprantami. Supaprastintai modeliuojant sistemą galima būtų išskirti šiuos etapus:

- Identifikuojame panaudojimo atvejus;
- Detalizuojame panaudojimo atvejį, pateikiant daugiau detalių kokiomis sąlygomis ir kaip ir kas vyksta;
- Iš panaudos atvejo detalizavimo šaltinio (19 pav.) išrenkamame daiktavardžius, tai kandidatai tapti klasėmis;
- Išrinkę veiksmažodžius, kurie jungią prieš tai išrinktus daiktavardžius, identifikuojame potencialius klasių metodus;
- Konsoliduojame išgrynintą informaciją, panaikiname pasikartojimus ir turime potencialias klases, jų metodus ir ryšius tarp klasių

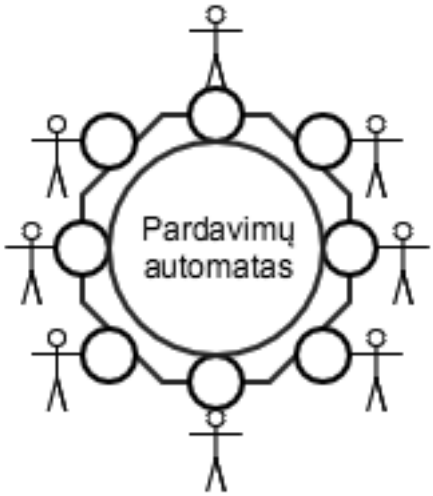


19 pav. Panaudojimo atvejų aprašymo galimybės. [ŠAF07]

Panagrinėkime trumpą pavyzdį, kurį pateikė Martin Logan vienos konferencijos metu [LOG14], tarkim mes norime sukurti pardavimo automato programą. Martin Logan yra Aktorių

modelio įgyvendinimo praktikas, daugelį metų dirbęs su Erlang programavimo kalba, bei knygos [LMC10] apie Erlang kalbos pritaikomumą lygiagretaus programavimo problemų sprendimams bendraautorius.

Vienas iš panaudojimo atvejų galėtų būti gėrimo išsirinkimas ir apmokėjimas už jį. Praplėčiant Martin Logan pavyzdį įvaizduokime, kad mūsų pardavimų automatas nėra standartinis, kur vienas automatas turi tik vieną vartotojo sąsają, mūsų pavyzdyje jis tebūnie turės 8 vartotojo sąsajas ir vienu metu galės aptarnauti iškart 8 vartotojus. Todėl toliau nagrinėjant *1 pavyzdį*, turėkime omeny, kad kiekvienas veiksmas ar funkcija, kurie paprastai gali būti atliekami vienos vartotojo sąsajos pardavimo automate, mūsų aptariamu atveju gali būti vykdomi lygiagrečiai aptarnaujant 8 skirtingus vartotojus, tokiu atveju kiekvienas veiksmas atliekamas ne vieną, o aštuonis kartus. Negana to, mes norime, kad vartotojas pirkdamas gėrimą, tai galėtų padaryti taip pat greitai tiek esant vienam, tiek aštuoniems vartotojams – esant didelei automato apkrovai vartotojo aptarnavimo greitis turi nenukentėti. Tai paprasta gyvenimiška situacija, kurią gali padėti spręsti Aktorių modelis.



20 pav. Pardavimų automatas su aštuoniomis vartotojo sąsajomis

Detalizuokime mūsų panaudojimo atvejį (*angl. use case*) pavyzdžiu:

1 pavyzdys:

Aš kaip vartotojas į pardavimo automatą sumetu pakankamą kiekį monetų ir spaudžiu pasirinkimo mygtuką, taip išsirenku norima gėrimą. Spaudžiu pirkimo mygtuką taip aktyvuoju automato mechanizmą, kuris išstumia mano pasirinktą gėrimą į prekių pasiėmimo dėklą. Gėrimas yra šaltas, nes pardavimo automato šaldymo sistema palaiko +4° C temperatūrą.

Taigi sekant aukščiau aprašytais žingsniais mes išrenkame visus unikalius daiktavardžius:

- ✓ *vartotojas,*
- ✓ *pardavimo automatas,*
- ✓ *monetos,*
- ✓ *pasirinkimo mygtukas,*
- ✓ *gėrimas, pirkimo mygtukas,*
- ✓ *automato mechanizmas,*
- ✓ *prekių pasiėmimo dėklas,*
- ✓ *šaldymo sistema,*
- ✓ *temperatūra*
- ✓ ..

- tai potencialūs programos objektai.

Dabar išrinkime veiksmažodžius, kurie jungia identifikuotos objektus:

- ✓ *sumesti,*
- ✓ *spausti,*
- ✓ *išsirinkti,*
- ✓ *aktyvuoti,*
- ✓ *išstumti,*
- ✓ *palaikyti*
- ✓ ..

– tai potencialūs objektų metodai;

Keletas objektų su savo metodais galėtų būti pvz.:

- ✓ *PasirinkimoMygtukas: spausti(),*
- ✓ *PirkimoMygtukas: spausti(),*
- ✓ *AutomatoMechanizmas: išstumpti(gėrimas)*
- ✓ ..

ir taip toliau.

Vėliau pritaikę kitus objektinio projektavimo principus, galėtume turėti sistemos modelį, kuris atitinka aprašytą panaudojimo atvejį, maždaug taip elgtumėmės modeliuodami objektais paremtą sistemą, tačiau kaip iš šios objektiniais projektavimo principais gautos informacijos galėtume galėtume identifikuoti Aktorius, visiškai neaišku.

Jau šiame etape suprantame, kad modeliuojant Aktorių modeliu paremtą sistemą, objekto sąvoka nelygi Aktoriaus sąvokai, nors kartais internete galima rasti minčių, kad tiesiog pavadinę objektus Aktoriais turėsime paruoštą modelį. Deja, kaip pamatysime toliau analizuodami galimus Aktorių identifikavimo būdus ne viskas taip paprasta.

Martin Logan 2014 metais „Lambda Jam“ konferencijos [LOG14] metu išreiškė mintį, kad siekiant supaprastinti Aktoriaus kaip primityvo vaizdinį, projektuojant sistemą Aktorių, kaip abstrakciją turėtume laikyti procesu. Pasinaudodami autoriaus išreikštomis mintimis pamėginkime rasti potencialius Aktorius dalykinės srities kontekste nagrinėjant tą patį pardavimo automato panaudojimo atvejį. Darykime prielaidą, kad Aktorius veikia panašiai kaip procesas. Taigi turime mąstyti, kaip suprojektuoti procesais paremtus sprendimus, tikėtina, kad šie procesai galėtų tapti kandidatais Aktorių modelio savybių pritaikymui.

Šiame etape, nėra didelio skirtumo, kokiomis priemonės vėliau bus įvykdyta techninė sprendimų realizacija, tai detalės prie kurių prieisime vėliau, tačiau dabar svarbus teisingas problemos išskaidymas, sąsajų tarp išskaidytų dalių suvokimas ir vykstančių procesų identifikavimas. Suprantant, kokias problemas gali padėti išspręsti Aktorius, svarbu identifikuoti tuos procesus, kurie gali būti vykdomi lygiagrečiai kitų procesų atžvilgiu.

Martin Logan [LOG14] teigia, kad ieškodami potencialių Aktorių, mus turėtų dominti, ne objektai, o procesai, todėl grįždami prie tolimesnio mūsų pavyzdžio nagrinėjimo, paanalizuokime, kaip iš panaudojimo atvejo išskirti procesus ir koks konkretaus proceso santykis su Aktoriumi.

6.4. Procesai ir Aktoriai

Kembridžo universiteto žodyno pateikiamas vienas iš proceso apibrėžimų „Atliekamų veiksmų ar įvykių visuma, turint tikslą pasiekti konkretų rezultatą“ [CDO15], tuo tarpu Carnegie Mellon universiteto programų sistemų inžinerijos institutas pažymi, kad procesas susideda iš šių komponentų[B07]:

- Rolių ir atsakomybių skirtų atlikti tam tikrą darbą;
- Įrankių ir padedančių tuos darbus atlikti;
- Procedūrų ir metodų apibūdinančių kaip užduotys turi būti atliekamos ir ryšių tarp užduočių;

Oksfordo universiteto žodynas [OD15] nurodo, kad procesas tai „Eilė veiksmų ir žingsnių su tikslu pasiekti konkrečią pabaigą“.

„Aktorius, tai yra fundamentalus skaičiavimo vienetas“ - tokį apibūdinimą pateikia pats Aktorių modelio kūrėjas Carl Hewitt [HMS12]. Skaičiavimas kompiuterijoje suprantamas, kaip veiksmas ar procesas atliekamas tam tikra seka ir laikantis tam tikrų taisyklių, pavyzdžiui algoritmų ir protokolų.

Remiantis aukščiau pateiktais procesų apibrėžimais, galime teikti, kad veiksmas gali būti traktuojamas, kaip proceso dalis arba sub-procesas. Iš to seka, kad Aktorius yra primityvus

skaičiavimo vienetas skirtas procesui arba jo daliai atlikti, siekiant konkretaus rezultato.

Žinodami, kokia prigimtinė Aktoriaus paskirtis, modeliuojant sistemą grįstą Aktorių modeliu, galėtume bandyti identifikuoti dalykinės srities procesus, kurių rezultatams pasiekti gali būti panaudoti skaičiavimo primityvai. Kitaip tariant tiksliniai procesai būtų tie, kurių rezultatas gali būti apskaičiuotas turint pradinius duomenis, o rezultatas pasiekiamas laikantis skaičiavimui apibrėžtų taisyklių (pvz. algoritmų).

Aktorius vienu metu gali atlikti vieną konkrečią užduotį, todėl identifikavus kompleksišką procesą, turėtume jį dekomponuoti į smulkesnes dalis, kurios vėliau taptų konkrečia užduotimi Aktoriui. Vienas konkretus Aktorius pats savyje neturi įgyvendinto išlygiagretinimo sprendimų, tačiau daug Aktorių veikiančių Aktorių sistemoje gali veikti lygiagrečiai, taip yra pasiekiamas išlygiagretinimo efektas išvengiant klaidų ir sudėtingų programinio kodo sprendimų, kuriuos aptarėme šio magistrinio darbo pradžioje. Konkrečioms užduotims pritaikytos išlygiagretimo schemos, tokios kaip aprašytos šiame darbe, įgalintų išnaudoti jau aptartas gerąsias Aktorių modelio savybes lygiagretaus programavimo kontekste.

6.4.1. Kaip identifikuoti procesus

Jei galvosime apie Aktorius, kaip apie lygiagretaus programavimo primityvus, turėtume galvoti kas mūsų modeliuojamoje sistemoje iš tiesų gali veikti lygiagretumu paremtais principais, akivaizdu, kad kalba turėtų eiti apie kažkokius veiksmus, taigi dar sykį patvirtiname, kad objektai, kaip kandidatai tapti Aktoriais čia iškrenta iš konteksto. Aptariami veiksmai turi turėti tiek pradžią tiek pabaigą. Šie veiksmai turėtų būtų tokie, kad jiems vykstant neprireiktų sprendimų, kurie paprastai naudojami tuomet, kai programos kodas vykdomas sinchroniškai, todėl blokavimosi mechanizmai turėtų nepatekti į mūsų aptariamą modelį. Taigi procesą galime pažinti, kaip logišką veiksmų seką, turinčią pradžią ir pabaigą.

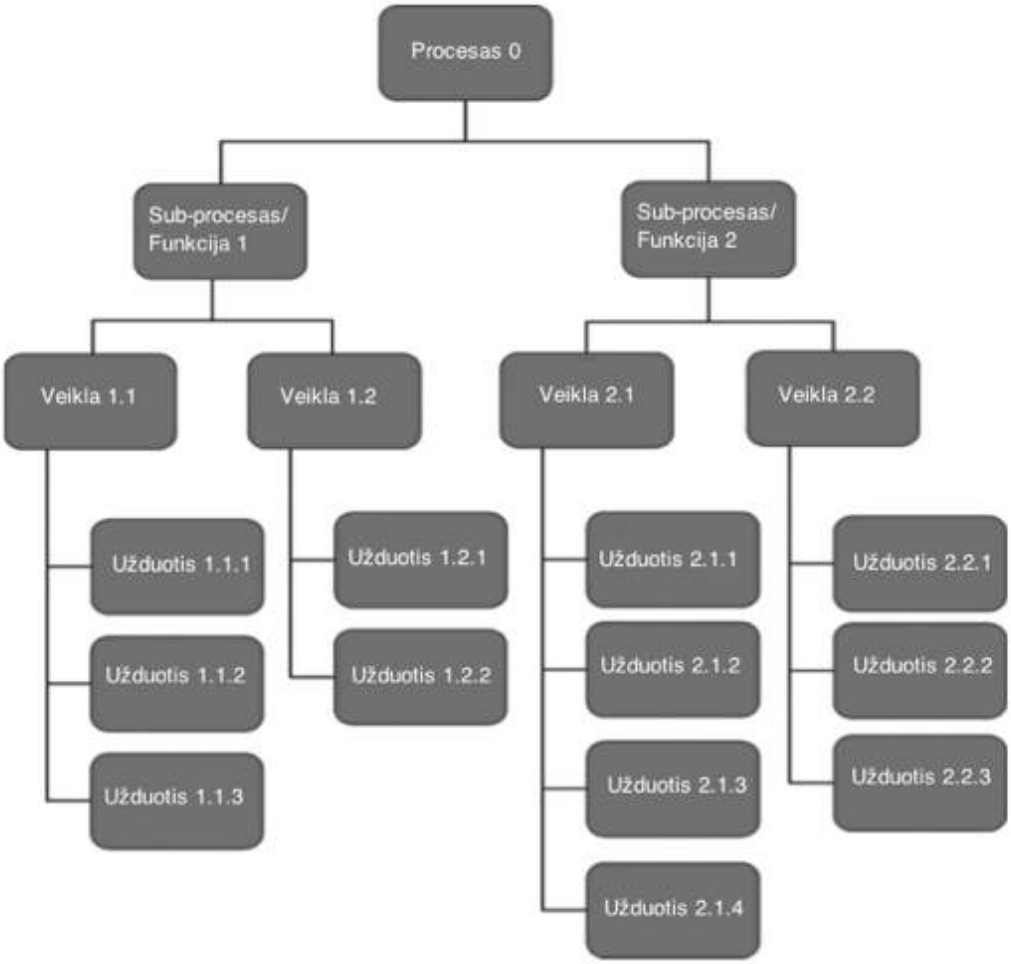
Iš tiesų matome, kad nagrinėdami *1 pavyzdį* objektinis projektavimas mums iki šiol davė ne daug naudos ir taip yra dėl paprastos priežasties, procesai nėra objektai. Procesą programinio kodo kontekste galėtume laikyti fundamentalia lygiagretumo esybe, nors dėl dalykinės srities konteksto apribojimų ne visi procesai vieni kitų atžvilgiu veikia lygiagrečiai, atskirų procesų realizacija Aktoriais kaip minėta neturi remtis blokavimusi, jei galima to išvengti.

Galvojant apie procesų išreiškimą Aktoriais mums netgi nelabai tinka turimos žinios apie gijų modeliavimo technikas, nes gijos Aktorių modelio kontekste suprantamos šiek tiek kitaip, nei klasikinio lygiagretaus programavimo atveju, kuris remiasi blokavimusi, sinchronizacija ir kitais elementais iš principo prieštaraujančiais Aktorių veikimo logikai. Remiantis Martin Logan idėja [LOG14] viskas apie ką mes turime galvoti projektuodami Aktorių modelio įgyvendinimą

sistemoje, tai procesai ir jų tarpusavio sąveika - kaip vieni procesai bendradarbiauja su kitais, bei šių procesų vidinius komponentus.

6.4.2. Procesų dekompozicija

Ankstesniuose skyriuose jau esame minėję, jog Aktorius vienu metu gali atlikti vieną užduotį, dėl šios priežasties procesai turėtų būti išskaidomi į smulkesnes dalis. Žinoma, jei procesas labai primityvus, nepasižymintis kompleksiškumu ir jo sudėtinės dalys niekaip negali būti išlygiagretintos, nauda tokio proceso įgyvendinimo Aktoriais būtų mažesnė, tačiau bendru atveju procesai dažnai turi daug ir sudėtingų sub-procesų, kurie taip pat turi atitinkamas sudedamąsias dalis, kurios gali būti vykdomos lygiagrečiai, todėl dekompozicija leidžia identifikuoti šias vietas.



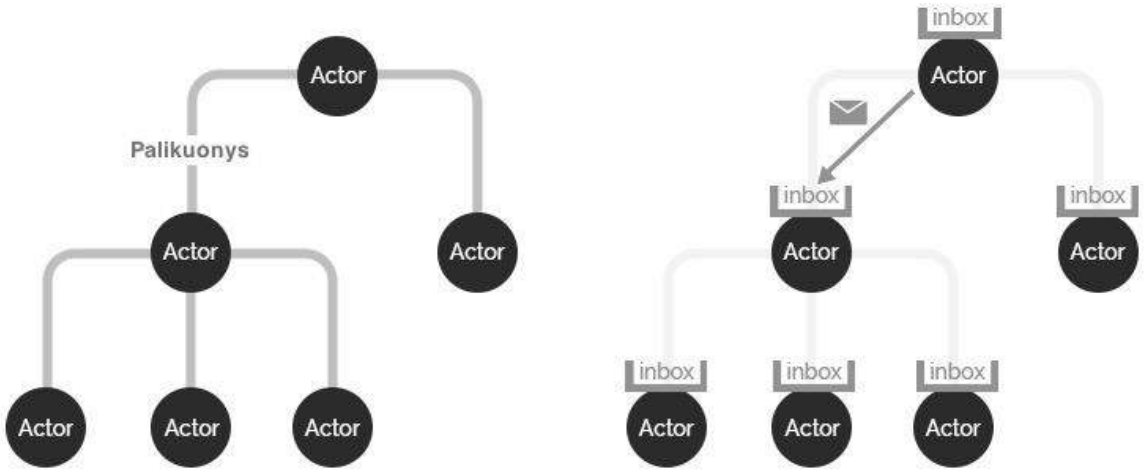
21 pav. Proceso hierarchija ir sudėtis

Procesas, kuris turi savyje kitų procesų vadinamas tėviniu (*angl. parent)* procesu. Procesas, kuris yra kito proceso vidinis procesas vadinamas vaikiniu (*angl. child*) procesu. Visi kartu vaikiniai procesai turi pilnai atspindėti visas tėvinio proceso veiklas.

Mūsų nedomina absoliučiai visi dalykinės srities kontekste vykstantys procesai, savaime suprantama, modeliuojant programų sistemą mus domina tik tie, kurie gali būti automatizuoti, kitaip tariant procesai, kurių bus gali būti atvaizduoti parašytoje programoje, todėl pagal nutylėjimą mes kalbame tik apie pastaruosius procesus. Pateiksime tris skirtingus metodus procesų dekompozicijai:

- *iš viršaus į apačią,;*
- *iš apačios į viršų;*
- *paremtus įvykiais.*

- **Iš viršaus į apačią** (angl. *Top-down*) – šiuo atveju pirmiausiai identifikuojame aukščiausio lygio procesus, vėliau identifikuojame vaikinius procesus, leidžiamės žemyn iki konkrečių veiklų, kol nebėra į ką skaidyti. Paprastai procesai, sub-procesai ir veiklos atsako į klausimą „kas yra atliekama?“, pasiekus konkrečios užduoties ir žingsnio stadiją klausimą „kas?“ pakeičia klausimas „kaip?“, tai vienas iš indikatorių apie dekompozicijos šakos pabaigą.
- **Iš apačios į viršų** (*angl. Button-up*) – atvirkštinis variantas iš viršaus į apačią būdui. Taikant šį metodą visų pirmą turime identifikuoti smulkiausias užduotis ir žingsnius, kuriuos reikia atlikti konkrečiam rezultatui pasiekti, vėliau pagal panašumo kriterijus atitinkamas veiklas grupuojame, jungiame į procesus. Šie procesai galimai taip pat grupuojami ir taip taip toliau, kol pasiekiama pilna procesų hierarchija.
- **Paremti įvykiais** (*angl. Event-driven*) – šiuo atveju turime galvoti apie trigerius, kuriems įvykus seka eilė veiksmų ir užduočių susijusių logine grandine.



22 pav. Aktorių hierarchinė struktūra [SØR15]

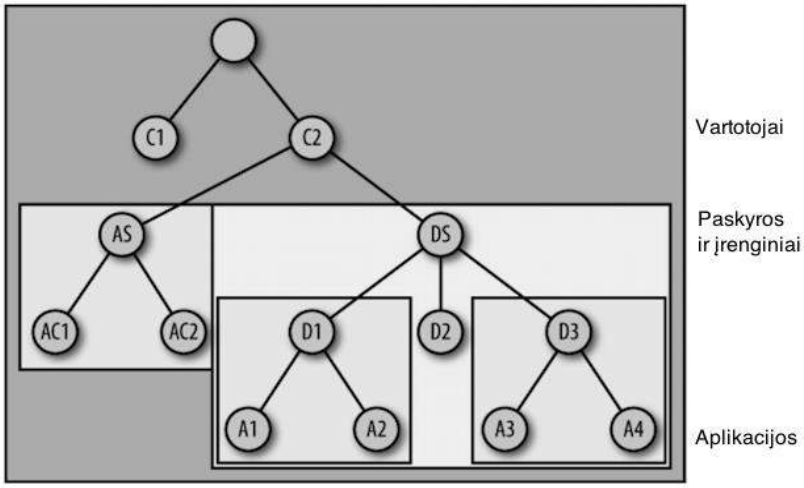
Kaip galime pamatyti iš 21 pav. ir 22 pav., kad tiek procesus tiek Aktorius galime atvaizduoti hierarchinėje medžio struktūroje. Dekomponavus procesų elementus, bei juos atvaizdavus medžio struktūroje, galėtume įžvelgti preliminarią Aktorių hierarchiją Aktorių sistemoje, kur kiekvienas Aktorius būtų atsakingas už tam tikros proceso dalies įgyvendinimą. Grįžkime prie mūsų nagrinėjamo panaudojimo atvejo (*1 pavyzdys)* ir išskirkime keletą procesų, kurie sąveikauja su kitais procesais, šie dar su kitais ir taip toliau:

- ✓ Monetų įdėjimas į pardavimo automatą sąveikauja su
- ✓ Įdėtų monetų apdorojimo procesu, šis sąveikauja su
- ✓ Kliento pasirinkimo apdorojimu, šis sąveikauja su
- ✓ Su vidinio mechanizmu procesu, kuris išstumia gėrimą į dėklą

Mes taip pat turime žemos temperatūros palaikymo procesą, tačiau matome, kad jis niekaip nesąveikauja su aukščiau išvardintais procesais. Šaldymo procesas vyksta visiškai nepriklausomai nuo šių procesų. Žvelgiant detaliau į pačių procesų nepriklausomumo lygį, pastebėsime, kad iš tiesų monetų įdėjimo į pardavimo automato procesas gali vykti nepriklausomai nuo to ar yra įgyvendintas koks nors monetų apdorojimo (pavyzdžiui leistinų monetų tikrinimo, monetų vertės apskaičiavimo) procesas, tuo tarpu monetų apdorojimo procesas gali vykti visiškai nesijaudinant ar kas nors pasirūpino prekės pasirinkimo apdorojimo procesu. Kitaip tariant procesai vyksta savarankiškai, jiems nereikia derintis vieniems prie kitų, net jei žinome, kad kartas nuo karto tenka tarpusavyje sąveikauti. Turint tokią informaciją apie procesus, jau galime pradėti mąstyti apie Aktoriais išreikštus sprendimus. Aktorius galėtų įgyvendinti kiekvieną iš tokių nepriklausomų procesų.

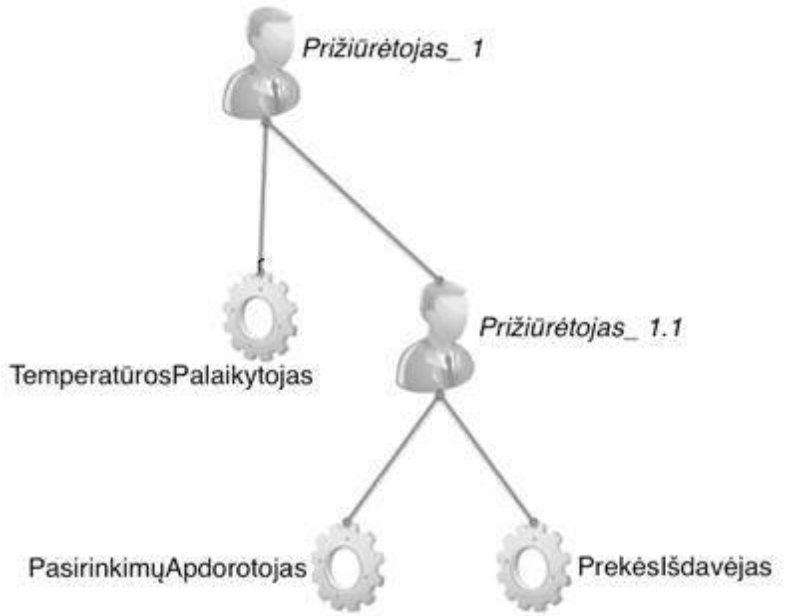
Aktorių modelio gerosios praktikos [ALL13] [VER15] sako, kad projektuojant sistemą, kuri, kaip taisyklė yra medžio tipo, turėtume apsibrėžti Aktorius Prižiūrėtojus (*angl. supervisor*). Jie

atsakingi už savo valdomų procesų sukūrimą ir priežiūrą. Prižiūrėtojas atsakingas už jam pavaldžios srities Aktorių veikimą, kurie pagal medžio struktūrą yra jo palikuonys (*angl. descendant*). Prižiūrėtojas esant klaidoms ar netipiškoms situacijoms gali perkurti Aktorių iš naujo ar atlikti kitus veiksmus siekiant užtikrinti nepertaukiamą sistemos veikimą. Taip mes sukuriame tolerantiškumo klaidoms mechanizmą (*angl. fault tolerance*).



23 pav. Sistemos priežiūra pagal trikių zonas [ALL13]

Taigi modeliuojant sistemos elementus, sukuriame šakninį sistemos pomedžio elementą - pomedžio Prižiūrėtoją, kuris vėliau bus įkomponuotas į bendrą sistemos medį, su visa savo pomedžio struktūra. Šioje pomedžio struktūroje žemiau gali būti kiti Prižiūrėtojai, kurie savo ruožtu prižiūri savo srities procesus. Aukščiausiai Prižiūrėtojo turėtų būti tie procesai, kurie neturi jokios sąveikos su kitais procesais esančiais tame pačiame pomedyje. Mūsų aptariamo pavyzdžio atveju, tai galėtų būti žemos temperatūros palaikymo procesas, kurį įgyvendintų Aktorius *TemperatūrosPalaikytojas*. Kaip atskirą šakninio Prižiūrėtojo vaikinį elementą, sukuriame kitą Prižiūrėtoją, kuris bus atsakingas už jo pomedyje esančius procesus ir taip toliau, kol visi procesai turės savo vietą medyje. Taigi modeliuojant pavyzdyje naudotą panaudojimo atvejį aptarti procesai galėtų būti įgyvendinti Aktoriais.(24 pav.)



24 pav. Aktoriai Prižiūrėtojai ir procesus įgyvendinantys Aktoriai[LOG14]

Taisyklės kurių reiktų laikytis kuriant vaikinius Aktorius:

- Kiekvienas Aktorius turi daryti vieną ir tik vieną darbą;
- Kai darbų yra labai daug, išeitis yra kurti vaikinius Aktorius, kurie šiuos darbus padeda atlikti;
- Vaikiniai Aktoriai naudojami atlikti atskiras smulkesnes darbų dalis, kurios tarpusavyje nepersidengia (*angl. separation of concerns*);
- Vaikinio Aktoriaus tėvinis elementas medžio struktūroje turėtų būti Prižiūrėtojas, kuris reaguotų įvykus nesėkmei vykdant programinį kodą;

Atrodytų viskas būtų gan gražu ir paprasta, tačiau vis dar lieka šiek tiek neapibrėžtumo. Ką turėtume laikyti procesu, juk procesas gali būti suprantamas ir kaip labai bendrinis, mūsų pateiktoje situacijoje (1 *pavyzdys*) toks bendrinis procesas galėtų būti pvz.: visas gėrimo įsigyjimo procesas. Taip jis savyje turi daug kitų procesų, galime juos pavadinti ir sub-procesais, tačiau kas gali atsakyti koks yra teisingas procesų išskaidymas, kiek smulkūs ir kiek abstraktūs jie gali ir turi būti, koks teisingas procesų skaičius turėtų egzistuoti dekomponuojant konkretų panaudojimo atvejį. Nuo to iš dalies priklauso atsakymas į klausimą, kiek Aktorių reiks tokių procesų įgyvendinimui.

Siekiant sumažinti neapibrėžtumą siūlytume susipažinti kiek kitokiu požiūriu į sistemos, kurioje veikia Aktoriai projektavimą, kuris mūsų manymu galėtų būti pritaikytas Aktorių

specifikai. Kalbame apie dalykinės srities modeliu paremto projektavimo (*angl. Domain-driven design*) naudojamas praktikas.

Procesų dekompozicija mums leidžia detaliau suprasti kokios dalykinės srities veiklos yra vykdomos, kokios funkcijos atliekamos ir kokie rezultatai turi būti pasiekti, kaip minėjome pasinaudojant procesų medžiu galime identifikuoti potencialių Aktorių kontūrus, tačiau nepaisant to paprastai konkrečius rezultatus galima pasiekti skirtingais būdais, skirtinga veiksmų seka ir taikant skirtingus požiūrio taškus. Siekiant atrasti daugiau atsakymų į klausimus būtų galima pasinaudoti dalykinės srities modeliu paremto projektavimo principais.

6.5. Aktorių identifikavimas pagal jų tipus

Ieškodami tolimesnių kriterijų Aktorių identifikavimui, panagrinėkime, kokie gali būti Aktorių tipai ir kokių tipų aplikacijose jie gali būti sutinkami. Kokiems panaudojimo atvejams geriausiai tiktų Aktorių modeliu paremti sprendimai, Jamie Allen kompanijos, kuri yra Scala programavimo kalbos kūrėja konsultacijų Direktorius, yra ilgametis Aktorių modeliu paremtų sistemų kūrėjas ir Scala programavimo kalbos praktikas [ALL13] pastebi, kad tai labai priklauso nuo konkretaus tikslo, kurį norima pasiekti, tačiau jei pageidaujama sukurti aplikaciją, kurios tinkamam veikimui būtų svarbūs tokie faktoriai, kaip atskirų aplikacijos funkcinių dalių vykdymo išlygiagretinimas, aplikacija pasižymėtų skaliuojamumu tiek tarp vidaus tiek tarp išoriškai nutolusių mazgų, bei būtų atspari trikiams - Aktoriai puikiai tinka šiems sprendimams. Jei prisiminsime šio darbo pradžioje aptartą reaktyvios sistemos sampratą, pastebėsime, kad autoriaus išvardintos savybės puikiai atitinka reaktyvių sistemų savybes. Taip, iki šiol mažai konkretumo, pamėginkite panagrinėti kas po tuo slepiasi.

Jamie Allen išskiria du Aktorių modeliu paremtus aplikacijų tipus:

- grįstus dalykinės sritis modeliu (*angl. Domain-driven*) ir
- darbų paskirstymu (*angl. work distribution*).

6.5.1. Dalykinės srities modeliais grįstas projektavimas (DDD)

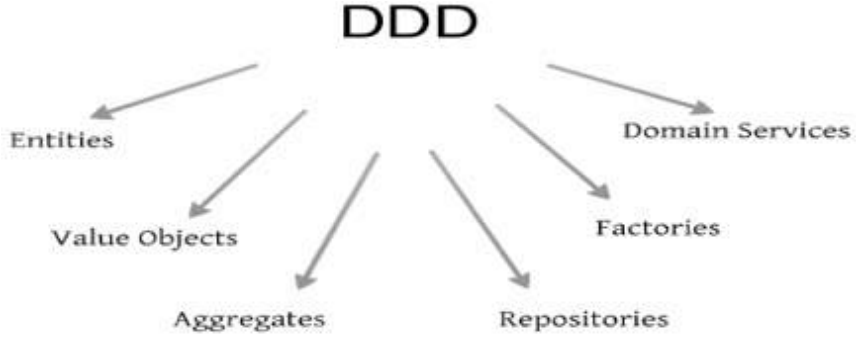
Dalykinės srities modeliu paremto projektavimo (*angl. Domain-driven design),* toliau – DDD, autorius Eric Evans 2003 metais knygoje „Domain-Driven Design: Tackling Complexity in the Heart of Software“ [EVA03] pateikė pagrindines idėjas ir technikas, kuriomis siūlo vadovautis siekiant kurti šiuolaikines, kompleksiškas programų sistemas. Vaughn Vernon glaudžiai bendradarbiaudamas su Eric Evans toliau dirbo ties DDD technikomis ir 2013 metais išleido knygą „Implementing Domain-Driven Design“ [VER13]. Vaughn Vernon kuris be to,

kad yra ilgametis programų sistemų kūrėjas ir projektuotojas ir architektas, yra vienas iš Reaktyvių sistemų, bei Aktorių modelio šalininkų ir praktikų. Vienas iš naujausių autoriaus darbų 2015 metais išleista knyga „Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka“ [VAR15], kurioje užsimena, kad Aktorių modelio paremtų programų sistemų projektavimas ir įgyvendinimas galėtų remtis DDD principais.

DDD nėra nei technologija nei metodologija, tai labiau kitoks požiūris į šiuolaikinių programų sistemų kūrimą. DDD nėra apie programinio kodo rašymą, pagrindinis fokusas skiriamas procesams ir žingsniams iki to, siekiant kad kuriama sistema maksimaliai atitiktų verslo funkcijas, galėtų būti lengvai palaikoma, būtų elastinga (*angl. elastic*) keičiantis sistemos apkrovimui, taigi ir skaliuojama (*angl. scalable*) - gebėtų prisitaikyti prie padidėjusio sistemos apkrovimo.

DDD esmė yra remiantis taisyklėmis, šablonais ir gerosiomis praktikomis identifikuoti dalykinės srities modelius, kurie reprezentuoja konkrečias verslo logikos dalis, jų komponavimas remiantis atitinkamomis taisyklėmis leidžia pasiekti aukščiau išvardintas programų sistemų savybes. Pats Eric Evans dalykinę sritį apibūdina kaip žinių, poveikio arba veiklos sritį „a sphere of knowlwdge, influence or activity“, o programų sistemos dalykinės srities objektą, kaip dalykinę sritį, kurios funkcijoms atlikti yra pritaikoma programa „The subject area to which the user applies a program is the domain of software “[EVA03].

Šio magistrinio darbo rėmuose mums aktuali DDD dalis tiek, kiek tai liečia Aktorių modelį, todėl detaliai visų DDD savybių ir praktikų nenagrinėsime. Identifikuojant, kuriant ir koreguojant dalykinės srities modelius DDD išskiriami 6 artifaktai: Esybės (*angl. Entities*), Vertės objektai (*angl. Value Objects*), Agregatai (*angl. Aggregates*), Repozitorijos (*angl. Repositories*), Fabrikai (*angl. Factories*), ir Dalykinės srities servisai (*angl. Domain Servises*) .



25 pav. Dalykinės srities modelis

6.5.2 Dalykinės srities modeliu grįsti Aktoriai

Jamie Allen [ALL13], taip kaip ir Vaughn Vernon [VER15], bei Andrew Easter [EAS13] sutinka, kad Aktoriai gali būti identifikuoti remiantis DDD - Dalykinės srities modeliais grįsto projektavimo principais *(angl. Domain-Driven Design).* Autorius pažymi, kad dalykine sritimi grįsti Aktoriai gyvena ir miršta, tam, kad atvaizduotų aplinkos būseną. Šiuo atveju Aktorius turi būsena, ši būsena atvaizduojama atmintyje (*angl. live cache*), o Aktorius kartu su būsena, kurią saugo atvaizduoja aplikacijos duomenis. Ši informacija gali būti sukurta ir prižiūrima laikantis jau anksčiau aptartos Aktorių hierarchinės struktūros 17 pav. Kalbant Dalykinės srities modeliais grįsto projektavimo sąvokomis šio tipo Aktoriai puikiai atitinka Eric Evans DDD paradigmoje naudojamus konceptus, tokius kaip Esybė (*angl. Entity*), Agregatas (*angl. Aggregate*) ir Šakninis agregatas (*angl. Aggregate Root*), kur paprastai tariant Aktorius įgyvendina Agregatą. Vaughn Vernon yra pateikęs rekomendacijas sudėtas į 3 atskirus darbus apie Agregato (taigi tuo pačiu ir Aktorių) identifikavimą ir teisingą projektavimą, išlaikant aukštu skaliuojamumu pasižyminčias savybes. [VER11-1] [VER11-2] [VER11-3].

Formuodami dalykinės srities modelio objektų hierarchiją išreikštą Aktoriais, turime galvoti kaip Aktoriai bus informuojami apie tai, kas vykstą aplinkoje, kurią reprezentuoja modelis ir kurioje jie egzistuoja, kad galėtų atlikti funkcijas, kurios jie yra sukurti. Kaip žinia, Aktoriai bendrauja siųsdami vieni kietiems žinutes, šiuo atveju gerosios praktikos sako [ALL13], kad šio tipo aplikacijose žinutės turėtų pranešti Aktoriui apie įvykius, kurie įvyko už modelio ribų. Dalykinės srities modelis turėtų reaguoti į išorės įvykius, keičiančius aplinką, kurią reprezentuoja atitinkamas modelis, Aktorius savo ruožtu būdamas informuotas apie atitinkamą įvykį reaguoja, taip, kad atspindėtų pasikeitusią situaciją modelyje.

6.5.3. Darbų paskirstymu grįsti Aktoriai

Pagrindinis skirtumas tarp Aktoriaus darbininko ir dalykinės srities modelio Aktoriaus yra tas, kad Aktorius darbininko funkcija yra atlikti su programos išlygiagretinimu susijusias užduotis, ir užduotis kurias dėl tam tikrų specifinių savybių, nori atskirti į atskirus blokus, šias užduotis Jamie Allen [ALL13] vadina pavojingomis, taigi Aktoriai darbininkai yra skirti būtent šioms funkcijoms atlikti. Tuo tarpu dalykinės srities modelio tipo Aktoriai, kurie turi būsena - atvaizduoja aplikacijos objektų būseną esamu momentu.

Šio tipo Aktoriai neturi būsenos (*angl. stateless*), jie gauna žinutes, kurios reprezentuoja tam tikrą objekto būseną, gavęs tokią žinutę Aktorius reaguoja atlikdamas numatytus veiksmus ir instrukcijas ir žinutės forma grąžina naują būseną reprezentuojančią informaciją.

Apibendrintai apie darbų paskirstymo Aktorius, galime pastebėti, kad Aktorius čia galėtume identifikuoti kaip šio darbo skyriuje *Išlygiagretinino schemos* aptartų schemų elementus, o jų hierarchinė struktūra būtų sąlygojama išlygiagretinimo schemose vykstančių procesų grandinės.

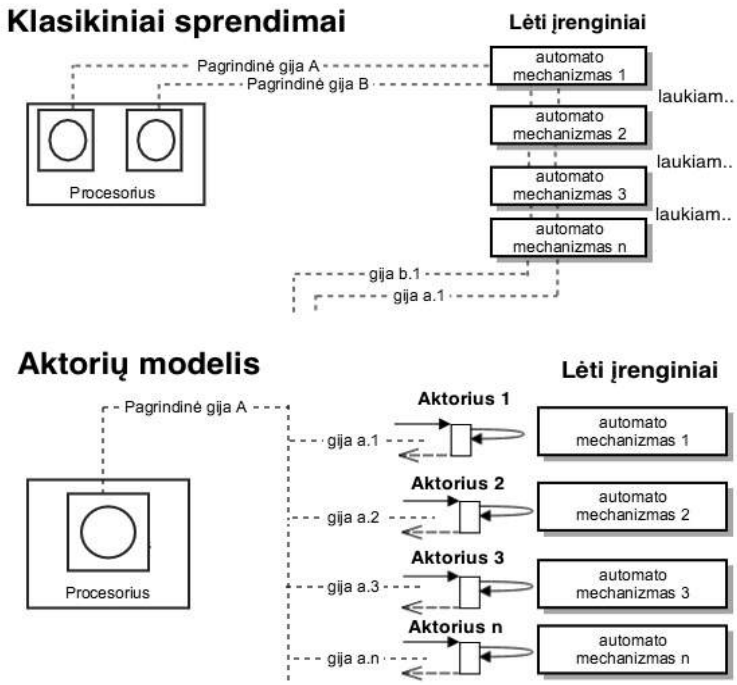
6.6. Aktoriai kaip sistemos greitinimo priemonė

Galėtume išskirti 2 pagrindines situacijas, kada Aktoriais grįsti sprendimai leistų užtikrinti spartesnį programų sistemos veikimą (*angl. speed-up*):

- Ilgai trunkančio algoritmo išlygiagretimas;
- Esant lėtiems įrenginiams;

- **Ilgai trunkančio algoritmo išlygiagretimas**
  Tais atvejais, kai uždavinys gali būti išskaidytas į tarpusavyje nesusijusias dalis, kurios yra atliekamos lygiagrečiai. Uždavinių išlygiagretinimo būdai tokie kaip *Šeimininko – tarno, Darbų krūvos, Konvejerio, Gamintojas-Vartotojas, bei Skaldyk ir valdyk* aptarti šio darbo ankstesniuose skyriuose. Lyginant su klasikinėmis priemonėmis Aktorių modelis leidžia sprendimus įgyvendinti paprasčiau, o turint pakankamai resursų padidinus Aktorių darbininkų kiekį ir greičiau.
- **Esant lėtiems įrenginiams**.
  Dėl lėto įrenginių veikimo paprastos gijos blokuojasi ir laukia savo eilės (21 pav.), kada galės tęsti darbą. Blokavimasis net tik iššaukia papildomą kompiuterio resursų poreikį, bet verčia ieškoti sudėtingų priemonių programuotojui.

  Aktorius, savo giją gauna tik tuomet, kai reikia atlikti konkretų darbą, o jei nieko nedaro, gija yra atiduodama kitam Aktoriui. Resursai visada yra riboti, o juos atlaisvinus efektyviai galima išnaudoti kitų užduočių atlikimui.



26 pav. Lėti įrenginiai sukelia blokavimąsi

Tiesa ta, kad vienas procesoriaus branduolys gali vykdyti vieną ir tik vieną užduotį vienu metu. Įsivaizduokime situaciją, kad dirbame kompiuteriu turinčiu vieną branduolį ir esame pasileidę daugiau nei vieną programą vienu metu: lygiagrečiai klausomės muzikos, archyvuojame failus, redaguojame dokumentus ar vykdome kitus įprastus darbus. Atrodytų, kad tuo pat metu vykdomas ne vienas procesas, o visa jų grupė, toks įspūdis susidaro, nes procesorius resursus tarp skirtingų užduočių paskirsto taip greitai, jog nepastebima jokių pertraukimų, todėl jei neįvyksta kažkokių neplanuotų sutrikimų viskas vyksta sklandžiai tarsi pertraukimų iš tiesų nebūtų. Situacija gali pasikeisti kuomet susiduriama su fiziniais įrenginiais, kurie yra prigimtinai lėtesni lyginant su kompiuterio vykdomų užduočių greičiais. Jei vykdant programinį kodą, reikia sulaukti išorinių įrenginių atsako, tam kad programą būtų galima vykdyti toliau, tai, kaip taisyklė, užtruks reikšmingą laiko momentą.

Įsivaizduokime jei mūsų nagrinėto *1 pavyzdžio* atveju, pardavimų automato programinė įranga būtų vykdoma vieno procesoriaus, turinčio vieną branduolį ir visi aštuoni vartotojai norėtų nusipirkti gėrimą vienu metu - turėtų prasidėti aštuoni skirtingi procesai. Kadangi pavyzdyje aptariamas pardavimų automatas savyje turi vidinių įrenginių, tokių kaip pinigų skaičiavimo mechanizmas ar mechanizmas, kuris išstumia gėrimą į atsiėmimo dėklą, vykdant kiekvieno kliento iniciuotus procesus susidurtume su situacija, kai tolimesnis programos vykdymas turėtų turėtų būti atidėtas kol nėra gautas atsakymas iš įrenginio. Kitaip tariant įvyktų atskirų programos sistemos dalių blokavimasis, su visomis iš to kylančiomis problemomis, kurios aptartos magistrinio darbo pradžioje. Tik gavus atsakymus iš įrenginių procesai galėtų

vykdomi toliau. Galiausiai galima ir tokia situacija, kad tam tikru momentu programinis kodas išvis nebus vykdomas, nes bus laukiama atsakymų iš lėtų įrenginių.

Šioje situacijoje mes matome dar vieną atvejį, kur galėtų sėkmingai pasitarnauti Aktoriai. Aktoriai nesiblokuoja, jei kiekvienam įrenginiui priskirtume už jį atsakingą Aktorių, kai tik bus gautas atsakymas iš įrenginio Aktorius apie tai būtų informuotas ir apie tai praneštų kitiems kitiems sistemos Aktoriams, kurie iškart galėtų pradėti vykdyti sekančias užduotis. Dėl asinchroninio komunikavimo ypatybės programa neturi blokuotis laukiant atsako iš lėtai veikiančio įrenginio. Aktorius laukia atsakymo, kol jis nieko nedaro, jam nėra priskirta resursų, tuo metu procesorius gali vykdyti kitas programinio kodo dalis, o prie šios užduoties grįžti tik tuomet kai jau bus turimi duomenys.

Kokie teigiami efektai yra pasiekiami:

- Procesorius vykdo kitas užduotis nesijaudindamas dėl Aktoriaus, kuris laukia įrenginio atsako. Aktorius pats apie save praneš, kai turės reikiamą informaciją tolimesniam darbų vykdymui;
- Maksimalus gijų skaičius yra ribotas, sistemoje reikšmingas užsiblokavusių gijų kiekis, gali sąlygoti resursų stygių naujoms gijoms kurti. Nors maksimalus Aktorių skaičius taip pat priklauso nuo procesoriaus pajėgumų ir turimos atminties, vienam Aktoriui reikia kur kas mažiau resursų nei vienai gijai. Sistema, kuri pajėgi palaikyti 5 000 gijų – yra pajėgi palaikyti net 1 200 000 Aktorių. [HO07];
- Gijos blokuojasi laukdamos lėtų įrenginių, didesnis sukurtų gijų skaičius reiškia, daugiau persijungimų skirstant turimus resursus. Procesoriaus resurso skirstymas tarp didelio kiekio gijų, taip pat reikalauja papildomų resursų;
- Programuotojas išvengia sudėtingo programinio kodo rašymo, siekiant užtikrinti deterministinį sistemos veikimą;
- Norint aptarnauti daug skirtingų vartotojų, sprendimus įgyvendinant Aktorių modeliu gali būti reikalingas mažesnis procesoriaus branduolių skaičius;
- Aktoriai leidžia pasiekti sistemos skaliuojamumą. Pridėjus daugiau resursų, įgalinama aptarnauti didesnį kiekį vartotojų.

Mes norėtume pažvelgti giliau ir nusileisti iki dar žemesnio lygio. Pamėginkime panagrinėti dar vieną paprastą pavyzdį, ir pažiūrėti ar programų sistemos elementams sąveikaujant su lėtais įrenginiais panaudojus Aktorių modelį mes galėtume su mažesniu resursų kiekiu aptarnauti daugiau vartotojų, nei klasikinių sprendimų atveju.

2 pavyzdys. Darbas su matricomis

Šimtas tūkstančių skirtingų vartotojų sistemai vienu metu paduoda Matricos A duomenis ir failo vardą, kuriame yra nurodyti Matricos B duomenys, sistema kiekvieno vartotojo Matricą A turi transponuoti, nuskaityti nurodytą failą diske, taip gauti Matricos B duomenis. Šias dvi Matricas sudauginti, galiausiai rezultatas turi būti įrašytas į failą.

***Matricos transponavimas**:*

$$[\mathbf{A}^{\mathrm{T}}]_{ij} = [\mathbf{A}]_{ji}$$

Jei **A** yra matrica $m \times n$, tai $\mathbf{A}^{\mathrm{T}}$ matrica $n \times m$.

Pavyzdžiui:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^{\mathrm{T}} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

Dvejų matricų daugyba:

$$c_{ij} = \sum_{k=1}^{s} a_{ik} b_{kj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \ldots + a_{is} b_{sj}$$

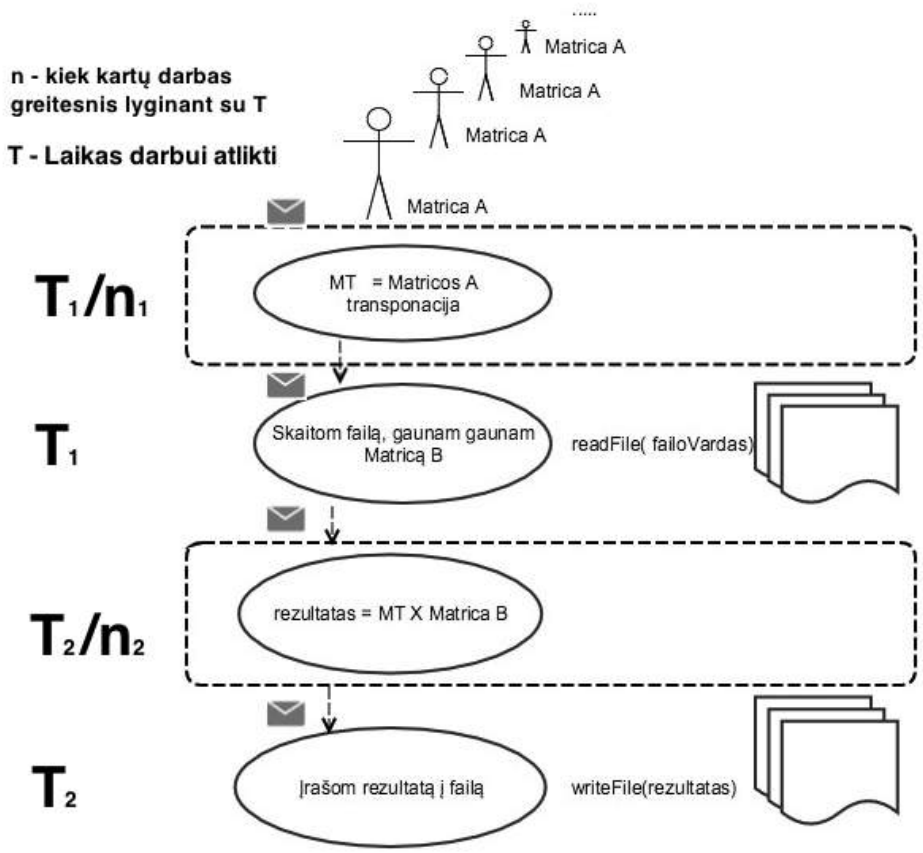
kur $1 \le i \le m$ ir $1 \le j \le n$.

Pavyzdžiui: $C = AB$, kai

$$A = \begin{bmatrix} 1 & 0 & 2 \\ -1 & 3 & 1 \end{bmatrix}, B = \begin{bmatrix} 3 & 1 \\ 2 & 1 \\ 1 & 0 \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 0 & 2 \\ -1 & 3 & 1 \end{bmatrix} \times \begin{bmatrix} 3 & 1 \\ 2 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} (1 \times 3 + 0 \times 2 + 2 \times 1) & (1 \times 1 + 0 \times 1 + 2 \times 0) \\ (-1 \times 3 + 3 \times 2 + 1 \times 1) & (-1 \times 1 + 3 \times 1 + 1 \times 0) \end{bmatrix} = \begin{bmatrix} 5 & 1 \\ 4 & 2 \end{bmatrix}$$

Galime pastebėti, kad šio panaudojimo atveju darbai išsiskiria į dvi dalis, tai tie darbai, kurie gali būti atlikti sąlyginai greitai, nes jų atlikimas nėra sąlygojamas lėtų įrenginių veikimu ir darbai, kurie yra priklausomi nuo įrenginių.



27 pav. Matricų uždavinys

Uždavinys, kuris iliustruotas 27 pav. gali būti sprendžiamas, bent keletu būdų. Vienas paprasčiausias, tačiau dėl savo brangumo nė nesvarstytinas duoti sistemai procesoriaus branduolių kiekį lygų vartotojų skaičiui, šiuo atveju akivaizdu, kad šimtas tūkstančių vartotojų būtų aptarnauti taip pat greitai, kaip vienas, nes kiekvienas vartotojas turėtų tik sau dedikuotą giją. Kitas variantas duoti procesoriaus branduolių kiekį gerokai mažesnį vartotojų skaičiui, tikintis, kad procesorius sukūręs daugiau gijų nei realiai yra branduolių sugebės tarp jų skirstyti resursus. Esant labai dideliam vartotojų skaičiui, kaip mūsų pavyzdyje, mes neišvengsime galybės vienu metu užsiblokavusių gijų, taigi procesoriaus reikalaujami pajėgumai vis dar būtų milžiniški. Trečias variantas galėtų būti patiems skirstyti gijas taip, kad sistema veiktų kur kas efektyviau nei pirmaisiais dviem atvejais.

Tiek matricos transponacija, kas reiškia eilučių sukeitimą vietomis su stulpeliais ir tiek abiejų matricų daugybos uždaviniai gali būti išlygiagretinti. Šiuo atveju, matytume kaip 2 atskirus procesus, kurie savyje turėtų savo sub-procesus. Už kiekvieną iš jų būtų atsakingas tėvinis Aktorius, kuris vadovaujantis vienu ar keliais šiame darbe pasiūlytais išlygiagretinimo būdais, perduotų užduotis Aktoriams darbininkams. Kitaip tariant abiem šiems uždaviniams

spręsti galėtume sukurti atitinkamas Aktorių hierarchijas, kur kiekvienas Aktoriaus egzempliorius būtų atsakingas už savo funkcijos atlikimą.
Failo skaitymo ir rašymo darbai tikėtina bus n kartų lėtesni lyginant su matricų transponacija ar matricų daugyba.

Pavyzdyje aprašyto uždavinio sprendimo etapus išskyrėme į procesų blokus. Kiekvieno bloko pagrindinis elementas Aktorius, gavęs asinchroninį pranešimą pradeda darbus, o turėdamas darbo rezultatą siunčia pranešimą kitam Aktoriui ir taip iki pat galo. Po kiekvienu iš šių blokų Aktorių slepiasi vaikinių Aktorių hierarchija (ji nepavaizduota 22 pav.), kurie savo ruoštu atliks jiems pavestas funkcijas. Ryšys tarp blokų nuoseklus, nors matricos transponacijos dalis galėtų būti vykdoma lygiagrečiai su matricos B duomenų gavimu.

Svarbiausia ką norėjome parodyti šiuo pavyzdžiu yra tai, kad darbų atlikimo laikas skirtinguose blokuose stipriai skirsis, tai mums perša išvadą, kad greiti darbai turi būti organizuojami kitokiais principais nei lėti. Manome, kad iš turimų procesoriaus resursų turėtume išskirti dalį, kuri bus alokuota greitiems darbams atlikti ir dalį palikti lėtesniems darbams, kurie šiuo atveju priklauso nuo lėto įrenginių veikimo. Jei galime apskaičiuoti, kad Matricos transponacija yra n kartų greitesnė, nei Matricos B duomenų gavimas, o abiejų matricų daugyba yra n kartų greitesnė už rezultato įrašymą į failus, greitesniuose blokuose esančių Aktorių naudojami procesoriaus resursai bus atlaisvinti per T/n laiką pilnai atlikus visų sistemos vartotojų pavestus darbus savo kuruojamoje srityje. Procesoriaus teikiamos gijos būtų naudojamos tik tuo momentu, kol Aktorius dirbs, vėliau būtų perleidžiamos kitoms sistemoms užduotims atlikti. Tai reiškia, kad tose sistemos dalyse, kuriose galime išlygiagretinti uždavinius pasinaudodami Aktorių modeliu mes galim sumažinti reikalaujamų resursų kiekį, nei spręsdami uždavinį sinchroninio komunikavimo priemonėmis.

Šiuo atveju mes galėtume pasinaudoti vienu iš Aktorių modelį įgyvendinančiu karkasu, pavyzdžiui AKKA, kuriame galima rasti įgyvendintus mechanizmus gijų paskirstymui tarp Aktorių, kuriuos galima valdyti. Paskutiniame šio magistro darbo etape mes bandysime įrodyti, kad toks resursų paskirstymas gali turėti įtakos programos vykdymo našumui.

6.7. Išvados dėl Aktorių identifikavimo

Išanalizavę metodus, kurie gali būti panaudoti Aktorių identifikavimui projektuojant programų sistemą ar atskiras jos dalis, norėtume pastebėti, kad nors Aktoriais galima įgyvendinti funkcinius reikalavimus, didžiausią Aktoriais grįstos sistemos vertę įžvelgiame kaip nefunkcinių reikalavimų įgyvendinimo būdą siekiant spartesnio sistemos veikimo, sistemos elastingumo, bei sistemos atsparumo sutrikimams;

Aktorių kiekis konkrečiam panaudojimo atvejui įgyvendinti gali skirtis priklausomai nuo konteksto ir savybių, kurias norima pasiekti, tačiau norėtume išskirti šiuos pagrindinius kriterijus, kuriais remiantis pateiksime savo išvadas:

- Procesai;
- Lėti įrenginiai;
- Lėti algoritmai, kurių dalis galima vykdyti lygiagrečiai;

- **Procesai.** Panaudojimo atvejų procesai ir jų dekompozicija, leidžia identifikuoti potencialią Aktorių hierarchiją. Vienas iš Aktorių kiekio panaudojimo atvejui nustatymo būdų, galėtų būti proceso hierarchijos medžio mazgų skaičius. Siekiant įgyvendinti kuo aukštesnį nepriklausomumo lygį tarp Aktorių, svarbu identifikuoti nepersikertančias procesų dalis, kuriuos galėtų būti vykdomos ne tik vieno konteksto ribose.
- **Lėti įrenginiai.** Manome, kad Aktorių kiekį didinantis veiksnys galėtų būti lėti įrenginiai, kurie sąlygoja vykdomos programos dalies blokavimąsi. Kiekvienam lėtam įrenginiui sistemoje galėtų būti dedikuotas Aktorius, kuris dėl asinchroninio komunikavimo savybių leistų efektyviau išnaudoti turimus kompiuterio resursus. Tuo tarpu neužsiblokavusios gijos galėtų būti nukreipiamos kitų sistemos darbų vykdymui.
- **Lėti algoritmai, kurių dalis galima vykdyti lygiagrečiai**. Jei uždavinys gali būti išskaidytas į tarpusavyje nesusijusias dalis, pritaikius išlygiagretinimo schemas Aktorių identifikavimas ir jų tarpusavio hierarchinė struktūra būtų sąlygojama išlygiagretinimo schemose vykstančių procesų grandinės.

LITERATŪROS SĄRAŠAS

[AGH86] G. Agha: Actors: a model of concurrent computation indistributed systems. MIT Press, Cambridge, MA, USA. ISBN 0-262-01092-5. 1986.

[ALL13] Jamie Allen: Effective Akka. O'Reilly Media. ISBN: 9781449360078. 2013.

[A13] J. Armstrong: Concurrent and parallel programming. 2013
[žiūrėta 2015-11-02]. Prieiga per internetą:
http://joearms.github.io/2013/04/05/concurrent-and-parallel-programming.html

[A03] J. Armstrong: Concurrency Oriented Programming in Erlang. Swedish Institute of Computer Science. 2003.
[žiūrėta 2015-10-16]. Prieiga per internetą:
http://www.guug.de/veranstaltungen/ffg2003/papers/ffg2003-armstrong.pdf

[B07] M. Bandor: Process and Procedure Definition: A Primer. Software Engineering Institute. Carnegie Mellon. 2007.
[žiūrėta 2015-10-16]. Prieiga per internetą:
http://sei.cmu.edu/library/assets/process-pro.pdf

[BLO13] Blogamith: Divide & Rule, the Akka way – Part – II. 2013.
[žiūrėta 2015-10-23]. Prieiga per internetą:
https://blogamith.wordpress.com/2013/11/28/divide-rule-the-akka-way-part-ii/

[BGT05] M. Brambilla, G. Guglielmetti, ir C. Tziviskou :Asynchronous Web Services Communication Patterns in Business Protocols. Politecnico di Milano, Dipartimento di Elettronica e Informazione. Milan, Italy. 2005.
[žiūrėta 2015-04-05]. Prieiga per internetą:
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.104.1376&rep=rep1&type=pdf

[BMOYTKWB13] J. Boner, E. Meijer, M. Odersky, G. Young, M. Thompson, R. Kuhn, J. Ward, G. Bort and others: The Reactive Manifesto. 2013.
[žiūrėta 2015-01-03]. Prieiga per internetą:
http://www.reactivemanifesto.org/pdf/the-reactive-manifesto.pdf

[CDO15] Cambridge Dictionaries Online. Process. 2015.
[žiūrėta 2015-01-03]. Prieiga per internetą:
http://dictionary.cambridge.org/us/dictionary/english/process

[CG88] N. Carriero, D. Gelernter: How to Write Parallel Programs. A Guide to the Perplexed. Yale University, Department of Computer Science, New Heaven, Connecticut. 1988.
[žiūrėta 2015-05-01]. Prieiga per internetą:
http://people.cs.aau.dk/~ask/Undervisning/MVP/html/p323-carriero.pdf

[EAS13] A. Easter: Akka, DDD, CQRS, Event Sourcing and Me. 2013. [žiūrėta 2015-11-07]. Prieiga per internetą: http://www.dreweaster.com/blog/2013/10/27/Akka-DDD-CQRS-Event-Sourcing-And-Me/

[ERB12] B. Erb: Concurrent Programming for Scalable Web Architectures . Institute of Distributed Systems Faculty of Engineering and Computer Science Ulm University. Diploma Thesis VS-D01-2012. 2012.

[EVA03] E. Evans: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison Wesley. ISBN 0-321-12521-5. 2003.

[HAL10] Ph. Haller: Isolated Actors for Race-Free Concurrent Programming. École Polytechnique Fédérale De Lausanne. Dissertation. THÈSE No 4874 (2010). Switzeland. 2010 [žiūrėta 2014-10-12]. Prieiga per internetą: http://infoscience.epfl.ch/record/151999/files/EPFL_TH4874.pdf

[HO08] Ph. Haller, M. Odersky, Scala actors: Unifying thread-based and event-based programming. Theoretical Computer Science. EPFL, Switzeland. 2008 [žiūrėta 2014-11-04]. Prieiga per internetą: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=A73B09C8AB060318775C01F552799621?doi=10.1.1.164.7047&rep=rep1&type=pdf

[HO06] Ph. Haller, M. Odersky, Event-Based Programming without Inversion of Control . Ecole Polytechnique Fédérale de Lausanne. 2006. [žiūrėta 2014-12-02]. Prieiga per internetą: http://lampwww.epfl.ch/~odersky/papers/jmlc06.pdf

[HO07] Ph. Haller, M. Odersky, Actors that Unify Threads and Events . In International Conference on Coordination Models and Languages, Ecole Polytechnique Fédérale de Lausanne. 2007. [žiūrėta 2014-12-02]. Prieiga per internetą: http://infoscience.epfl.ch/record/99729/files/haller07actorsunify.pdf

[HBS73] C. Hewitt Carl, P. Bishop and R. Steiger: A universal modular ACTOR formalism for artificial intelligence, in: Proceedings of the 3rd international joint conference on Artificial intelligence, IJCAI'73, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA

[HB78] C. Hewitt, H. Baker, Henry: Actors and Continuous Functionals. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA. 1978.

[HEW15] C. Hewitt: Actor Model of Computation: Scalable Robust Information Systems, CU'73, Ithaca, NY, USA [žiūrėta 2015-02-01]. Prieiga per internetą: http://arxiv.org/pdf/1008.1459v38.pdf

[HP85] D. Harel, A. Pnueli. On the development of reactive systems. Logics and Models of Concurrent Systems, NATO ASI Series. 1985. [žiūrėta 2015-04-09]. Prieiga per internetą: http://www.wisdom.weizmann.ac.il/~harel/SCANNED.PAPERS/ReactiveSystems.pdf

[HMS12] C.Hewitt, E. Meijer, C. Shypersky. The Actor Model (everything you wanted to know, but were afraid to ask). 2012.
[žiūrėta 2015-04-09]. Prieiga per internetą:
http://channel9.msdn.com/Shows/Going+Deep/Hewitt-Meijer-and-Szyperski-The-Actor-Model-everything-you-wanted-to-know-but-were-afraid-to-ask

[KA11] R. K. Karmani, G. Agha: Actors. Open Systems Laboratory Department of Computer Science University of Illinois at Urbana-Champaign. Encyclopedia of Parallel Computing. 2011.

[KSA09] R. K. Karmani, A. Shali, G. Agha: Actor Frameworks for the JVM Platform: A Comparative Analysis. University of Illinois at Urbana-Champaign. 2009.
[žiūrėta 2015-09-10]. Prieiga per internetą:
http://osl.cs.illinois.edu/media/papers/karmani-2009-pppj-actor_frameworks_for_the_jvm_platform.pdf

[LOG14] M. Logan: Design and architecture for actors. Conference Lambda Jam 2014. Chicago. 2014.
[žiūrėta 2015-10-15]. Prieiga per internetą:
https://www.youtube.com/watch?v=8AXec4l0UL4

[LMC10] M. Logan, E. Merritt, R. Carlsson: Erlang and OTP in Action. Manning. ISBN 9781933988788. 2010.

[MAC97] P. Mackay: Why has the actor model not succeeded? ICSTM'97, London, United Kingtom
[žiūrėta 2015-04-01]. Prieiga per internetą:
http://www.doc.ic.ac.uk/~nd/surprise_97/journal/vol2/pjm2/

[M015] P. Matteti: Akka with Scala. Wright State University. 2015
[žiūrėta 2015-11-05]. Prieiga per internetą:
http://cecs.wright.edu/~pmateti/Courses/7370/Lectures/Actors+Akka+Scala/akka.html

[McK15] P. E. McKeney: Is Parallel Programming Hard, And, If So, What Can You Do About It? Linux Technology Center. 2015

[N11] B. Nobakht: Multicore Programming in Object-Oriented Languages. Leiden Institute of Advanced Computer Science. Leiden University. 2011.
[žiūrėta 2015-10-01]. Prieiga per internetą:
http://narmnevis.com/wp-content/uploads/2011/03/Multicore_Programming_Object_Oriented_Languages.pdf

[OD15] Oxford Dictionaries. Process. 2015
[žiūrėta 2015-10-16]. Prieiga per internetą:
http://www.oxforddictionaries.com/us/definition/american_english/process

[P96] C.M. Pancake: Is Parallelism for You? Oregon State University. Originally published in Computational Science and Engineering, Vol. 3, No. 2. 1996.
[žiūrėta 2015-01-05]. Prieiga per internetą:
http://web.engr.oregonstate.edu/~pancake/papers/IsParall/

[PAR15] Parallel computing. Straipsnių rinkinys. 2015
[žiūrėta 2015-11-24]. Prieiga per internetą:
http://www.cse.unt.edu/~tarau/teaching/parpro/papers/Parallel%20computing.pdf

[SJ13] S. Sudhakar, R. Jain: The Need for Speed: Understanding Design Factors that Make Multi-core Parallel Simulations Efficient. Mentor Grafics. 2013.
[žiūrėta 2014-10-03]. Prieiga per internetą:
https://verificationacademy.com/verification-horizons/june-2013-volume-9-issue-2/The-Need-for-Speed-Understanding-Design-Factors-that-Make-Multi-core-Parallel-Simulations-Efficient

[ShS12] Sh. Imam V. Sarkar: Integrating Task Parallelism with Actors, Rice University. 2012.
[žiūrėta 2015-02-15]. Prieiga per internetą:
http://shams.web.rice.edu/papers/2012-oopsla-actors.pdf

[OA06] J. L. Ortega Arjona: Architectural Patterns for Parallel Programming: Models for performance Estimation. Department of Computer Science University College London. Dissertation. 2006.
[žiūrėta 2015-05-12]. Prieiga per internetą:
http://www.sigsoft.org/phdDissertations/theses/JorgeOrtega.pdf

[SØR15] C. Sørensen: An Actor model example with akka.net. 2015.
[žiūrėta 2015-12-27]. Prieiga per internetą:
http://blog.geist.no/an-actor-model-example-with-akka-net/

[ŠAF07] P. Šafranauskas: Veiklos modelio ir vartotojo reikalavimų (Use-Case) modelio sąsajos tyrimas. Magistro darbas. Kauno technologijos universitetas. 2007.
[žiūrėta 2015-12-07]. Prieiga per internetą:
http://vddb.library.lt/fedora/get/LT-eLABa-0001:E.02~2007~D_20070816_143800-45511/DS.005.0.02.ETD

[VAI07] R. Vaicekauskas. Lygiagretūs ir išskirstyti skaičiavimai. Mokymo medžiaga. Vilnius, 2007.

[VER11-1] V. Vernon: Effective Aggregate Design Part I: Modeling a Single Aggregate. 2011.
[žiūrėta 2015-11-12]. Prieiga per internetą:
https://vaughnvernon.co/wordpress/wp-content/uploads/2014/10/DDD_COMMUNITY_ESSAY_AGGREGATES_PART_1.pdf

[VER11-1] V. Vernon: Effective Aggregate Design Part I: Modeling a Single Aggregate. 2011.
[žiūrėta 2015-11-12]. Prieiga per internetą:
https://vaughnvernon.co/wordpress/wp-content/uploads/2014/10/DDD_COMMUNITY_ESSAY_AGGREGATES_PART_1.pdf

[VER11-2] V. Vernon: Effective Aggregate Design Part II: Making Aggregates Work Together. 2011.
[žiūrėta 2015-11-12]. Prieiga per internetą:
https://vaughnvernon.co/wordpress/wp-content/uploads/2014/10/DDD_COMMUNITY_ESSAY_AGGREGATES_PART_2.pdf

[VER11-3] V. Vernon: Effective Aggregate Design Part III: Gaining Insight Through Discovery. 2011.
[žiūrėta 2015-11-12]. Prieiga per internetą:
https://vaughnvernon.co/wordpress/wp-content/uploads/2014/10/DDD_COMMUNITY_ESSAY_AGGREGATES_PART_3.pdf

[VER15] V. Vernon: Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka. Addison-Wesley Professional. ISBN 9780133846904. 2015.

[VER13] V. Vernon: Implementing Domain-Driven Design. Addison-Wesley Professional. ISBN 9780133039900. 2013