VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
PROGRAMŲ SISTEMŲ KATEDRA

Išskirstytų sistemų technologijų tyrimas

Investigation of Distributed Systems Technologies

Magistro baigiamasis darbas

Atliko:	Donatas Žąsinas	(parašas)
Darbo vadovas:	Doc. dr. Sigitas Dapkūnas	(parašas)
Recenzentas:	Lekt. Tadas Savičius	(parašas)

Vilnius – 2009

Santrauka

Šiame darbe nagrinėjamos išskirstytos sistemos ir jų kūrimui naudojamos technologijos. Kadangi technologijų yra daug, o visas jas ištirti ir palyginti būtų labai sunku, todėl didžiausias dėmesys skiriamas Java RMI ir .Net Remoting – tai dvi populiarios išskirstytų objektų paradigmos šiuo metu naudojamos programų sistemų kūrimui Pagrindinis darbo tikslas yra išanalizuoti ir visapusiškai palyginti pasirinktas technologijas, didelį dėmesį skiriant jų veikimo greičiui. Darbas sudarytas iš keturių dalių. Pirmoje dalyje pateikiama bendra informacija apie išskirstytas sistemas, jų vystymąsi, architektūras, technologijų raidą ir t.t.. Antroje dalyje nagrinėjama Java RMI ir .Net Remoting veikimo principai, kuriais remiasi šios technologijos. Trečioje dalyje pateikiami šių dviejų technologijų veikimo principų panašumai ir skirtumai. Ir galiausiai siekiant išsiaiškinti Java RMI ar .Net Remoting realiomis sąlygomis veikia greičiau, ketvirtoje dalyje pateikiamas atlikto vykdymo greičio tyrimo aprašymas. Atlikti darbai leidžia teigti, kad Java RMI ir .Net Remoting idėjiškai yra panašios technologijos, bet skirtumų, įtakojančių veikimą, tarp jų irgi yra. Greičio tyrimo metu gauti rezultatai parodė, kad .Net Remoting beveik visais atvejais yra lėtesnė nei Java RMI.

Raktiniai žodžiai: išskirstytos sistemos, Java RMI, .Net Remoting

Summary

Distributed systems and technologies used for this type of software are analyzed in this paper. Since there are a lot of technologies and it would be very difficult to explore and compare all of them, therefore this paper mainly focuses on Java and RMI. Net Remoting - two popular distributed objects paradigms currently used in distributed systems development. The first part provides general information about distributed systems, their development, architectures, technologies, and so on. The second part of the paper analyses the operating principles of Java RMI and. Net Remoting. The third section determines technological similarities and differencies. And finally, in order to clarify the fact Java RMI or. Net Remoting works faster in real terms, the fourth part provides the results of the implemented speed test. Although Java RMI and. Net Remoting are similar technologies, but the analysis showed that the there are quite a lot differences between them. The speed survey showed that. Net Remoting is slower than Java RMI in almost all cases.

Keywords: distributed systems, Java RMI, .Net Remoting

Turinys

Įvadas ... 6
1. Išskirstytų sistemų principai ir technologijų raida ... 8
  1.1. Išskirstytos architektūros ... 9
    1.1.1. Modulinis programavimas ... 9
    1.1.2. Klientas-Serveris ... 9
    1.1.3. N Lygių (N-Tier) ... 9
    1.1.4. Lygus su lygiu (Peer-to-Peer) ... 11
  1.2. Išskirstytos technologijos ... 12
    1.2.1. Prievadai (Sockets) ... 13
    1.2.2. Nutolusių procedūrų kvietimai ... 13
    1.2.3. Išskirstyti objektai ... 14
    1.2.4. DCE/RPC ... 15
    1.2.4. CORBA ... 15
    1.2.4. DCOM ... 16
    1.2.5. MTS/COM+ ... 16
    1.2.6. Java RMI ... 16
    1.2.7. Java EJB ... 17
    1.2.8. Web Servisai/SOAP/XML-RPC ... 17
    1.2.9. .Net Remoting ... 18
2. Technologijų principų analizė ... 18
  2.1. Java RMI ... 18
    2.1.1. Paprastos išskirstytos sistemos pavyzdys naudojant Java RMI ... 19
    2.1.2. Duomenų perdavimas ... 21
    2.1.3. Laido protokolas, RMI-IIOP ... 21
    2.1.4. Vardų paslaugos ... 22
  2.2. .Net Remoting ... 22
    2.2.1. Paprastos išskirstytos sistemos pavyzdys naudojant .Net Remoting ... 23
    2.2.2. Duomenų perdavimas ... 25
    2.2.3. HTTP kanalas ir TCP kanalas ... 25
3. Java RMI ir .Net Remoting veikimo principų palyginimas ... 26
  3.1. Panašumai ... 27
    3.1.1. Objektiškai orientuoti RPC ... 27
    3.1.2. Interfeiso apibrėžimo kalba ... 27
    3.1.3. Serverio objektų gyvavimo ciklas ... 27
  3.2. Skirtumai ... 27
    3.2.1. Nutolusių objektų nuorodų saugojimas ... 27
    3.2.2. Nutolusių objektų deklaravimas ir realizavimas ... 28
    3.2.3. Programavimo lengvumas ... 28
    3.2.4. Skirtingų kalbų ir platformų sąveika ... 28
    3.2.5. Aktyvavimas ... 29
    3.2.6. Kiti skirtumai ... 29
4. Java RMI ir .Net Remoting veikimo greičio tyrimas ... 29

4.1. Matavimų specifikacija.....29
4.2. Testavimo aplinka.....30
4.3. Matavimams naudotų programų realizacijų ypatumai.....31
4.4. Matavimų rezultatai.....32
4.4.1. Vieno kliento scenarijus.....32
4.4.1.1. Paprasti duomenų tipai.....32
4.4.1.2. Skirtingi simbolių eilutės ilgiai.....34
4.4.1.3. Skirtingo dydžio tipizuoti sąrašai.....36
4.4.2. Kelių klientų scenarijus.....38
4.4.2.1. Paprasti duomenų tipai.....39
4.4.2.2. Skirtingi simbolių eilutės ilgiai.....40
4.5. Papildomi tyrimai.....42
Rezultatai ir išvados.....43
Šaltinių sąrašas.....45
Priedai.....46

Įvadas

Išskirstytų sistemų sritis buvo plėtojama jau prieš dvidešimt, trisdešimt metų, o ypač spartus interneto ir intraneto tinklų plitimas bei tobulėjimas, per paskutinį dešimtmetį padarė šią sritį šiuolaikinės programinės įrangos kūrimo pagrindu. Kuo toliau tuo labiau vystosi ir populiarėja įvairūs mobilieji įrenginiai, tokie kaip delniniai kompiuteriai, mobilūs telefonai, nešiojami kompiuteriai ar kiti specializuoti mobilūs įrenginiai, o kartu su jais ir daugybė gerai išvystytų komunikacijos būdų, tokių kaip GSM tinklai, bevieliai ir paprasti kompiuteriniai tinklai, beveik visur prieinamas internetas ar net palydovinis ryšys. Todėl šiandien vienu ar kitu būdu beveik visur susiduriama su išskirstytomis sistemomis, nes jos įgyvendina tai kas yra aktualiausia verslui, mokslui ir net paprastam žmogui. Šiandien išskirstytos sistemos yra visur: nuo paprastesnių internetinių taikomųjų programų ar į tinklą sujungtų kompiuterių šiuolaikiniame automobilyje iki integruotų bankinių sistemų ar sudėtingų išskirstytų skaičiavimo tinklų [JZ05].

Šiai dienai rinkoje yra daug uždarų (vidinių) ar atviro standarto išskirstytų sistemų technologijų, pavyzdžiui: Java RMI, CORBA, .Net Remoting, DCOM. Visos šios išvardintos technologijos yra komponentinės, tai reiškia, kad programos yra sudarytos iš komponentų rinkinių, kurie yra išskirstyti įrenginių tinkle ir kartu veikia kaip vienos didelės programų sistemos dalys [PRP06]. Kiekviena iš šių technologijų turi savų pliusų bei minusų. Ir nors kasdieniame gyvenime mes dažnai matome ir susiduriame su išskirstytomis sistemomis, tačiau dažniausiai apie tai kaip jos veikia nė nesusimastome. Jei reikėtų patiems sukurti programą, kuri bendrauja su nutolusiais kompiuteriais (įrenginiais), susidurtumėme su daugybe klausimų – kokios yra galimybės, kaip viską sujungti į vieną visumą, kokios egzistuoja technologijos ir kurią iš jų reikėtų pasirinkti pagal turimus sistemos reikalavimus ir charakteristikas? Neretai tai būna gana sudėtingas sprendimas.

Kadangi technologijų yra daug, o visas jas ištirti ir palyginti būtų labai sunku, šiame darbe didžiausias dėmesys bus skirtas Java RMI ir .Net Remoting – tai dvi populiarios išskirstytų objektų paradigmos šiuo metu naudojamos programų sistemų kūrimui. Tiek Java RMI tiek .Net Remoting leidžia programuotojams kurti išskirstytas sistemas, per daug nesirūpinant žemo lygio išskirstytų sistemų programavimo problemų detalėmis. Naudojant šias technologijas išskirstytoje objektinėje sistemoje, objektai veikiantys skirtinguose kompiuteriuose programai atrodo kaip vietiniai. Iš pirmo žvilgsnio gali pasirodyti, kad šios technologijos yra visiškai vienodos, tik skirtos skirtingoms platformoms, bet kaip yra iš tikrųjų? Kuo jos panašios ir kuo skiriasi? Kuri technologija yra geresnė?

Pagrindinis šio, iš keturių dalių susidedančio, darbo tikslas - išanalizuoti ir visapusiškai palyginti pasirinktas išskirstytų sistemų kūrimui naudojamas technologijas, didelį dėmesį skiriant jų veikimo greičiui. Kad labiau įsigilintumėme į šią dalykinę sritį pirmoje dalyje bus pateikiama

bendra informacija apie išskirstytas sistemas, jų vystymąsi, architektūras, technologijų raidą ir t.t.. Antroje dalyje išnagrinėsime Java RMI ir .Net Remoting principus, kuriais remiasi šios technologijos. Pateiksime po paprastos išskirstytos sistemos pavyzdį, kad būtų aišku, kokie pagrindiniai žingsniai reikalingi tokiai sistemai sukurti. Trečioje dalyje išanalizuosime šių dviejų technologijų veikimo principų panašumus ir skirtumus, pabandysime nustatyti, kuri iš jų atrodo pranašesnė. Ir galiausiai siekiant išsiaiškinti Java RMI ar .Net Remoting realiomis sąlygomis veikia greičiau, ketvirtoje dalyje bus pateiktas atlikto vykdymo greičio tyrimo aprašymas, gauti rezultatai, jų analizė bei įvertinimas.

1. Išskirstytų sistemų principai ir technologijų raida

Išskirstyta sistema – tai sistema susidedanti iš autonomiškų kompiuterių, sujungtų komunikacijos tinklais, ir programinės įrangos, kurios suprojektavimo ir realizavimo ypatumai įgalina šią terpę veikti kaip vieningą, integruotą sistemą. Išskirstytų sistemų dėka žmonės gali greičiau ir efektyviau atlikti sudėtingas užduotis, bendradarbiauti ir koordinuoti savo veiksmus. Pagrindiniai privalumai gaunami kuriant įvairias išskirstytas sistemas yra šie: dalinimasis resursais, atvirumas, lygiagretumas, plečiamumas ir klaidų tolerancija [JZ05].

- *Dalinimasis resursais*. Išskirstytoje sistemoje resursai, tokie kaip kompiuterinė įranga, programinė įranga ar konkretūs duomenys gali būti lengvai visiems prieinami (pavyzdžiui nedideli kompiuteriai, kurie neturi pakankamai veikimo galios, kad susitvarkytų su dideliais duomenų kiekiais ar sudėtingais skaičiavimais, intensyvaus darbo reikalaujančius skaičiavimus galėtų perleisti galingam serveriui arba įgyvendinus lygus su lygiu (peer-to-peer) architektūros tipo programą, atliekamo darbo krūvį dalintųsi visi klientai).
- *Atvirumas*. Išskirstytų sistemų atvirumas gaunamas paviešinant programinės įrangos interfeisą ir padarant jį prieinamu kūrėjams. Tokiu būdu galima lengvai integruoti tokios sistemos teikiamas paslaugas, nes naudojami standartizuoti protokolai.
- *Lygiagretumas*. Vykdymo lygiagretumą (dažniausiai išskirstytiems skaičiavimas skirtose sistemose) galima pasiekti siunčiant užklausas keliems į tinklą pajungtiems kompiuteriams iš karto.
- *Plečiamumas*. Išskirstyta sistema įdiegta nedideliame kompiuterių kiekyje gali būti lengvai praplėsta ir įdiegta į daugiau kompiuterių taip padidinant visos sistemos veikimo galimybes ir našumą (nuo paprastų Klientas-Serveris sistemų iki skaičiavimo tinklų (Cluster, GRID)).
- *Klaidų tolerancija*. Kompiuteriai sujungti tinklu gali būti kaip atsarginiai resursai. Ta pati programinė įranga gali būti įdiegta keliuose kompiuteriuose todėl net ir atsiradus kompiuterinės ar programinės įrangos sutrikimams, klaidos gali būti aptinkamos ir darbas atliekamas panaudojant atsarginius resursus.

Jei išskirstyta sistema ar viena iš jos dalių bus suprojektuota ir sukurta netinkamai, tai iš pažiūros net ir menka klaida gali sumažinti sistemos efektyvumą, iškreipti gaunamus rezultatus arba sutrikdyti kitas sistemos dalis ir taip galutinai sugriauti visos sistemos darbą. Gedimų lokalizavimo ir diagnozavimo problemos išskirstytoje sistemoje taip pat gali tapti sudėtingesniu uždaviniu nei paprastoje programoje, nes sutrikimas gali būti bet kurioje sistemos dalyje, skirtingoje aplinkoje ir gal net operacinėje sistemoje, todėl situacijai išsiaiškinti gali prireikti

išsamios nutolusių mazgų ar ryšių tarp jų analizės bei testavimo [MNW03]. Išskirstytos sistemos kuriamos naudojant įvairias, daugelį metų besivysčiusias, technologijas ir architektūras, kas taip pat turi daug įtakos bendram sistemos funkcionavimui, juolab kai susiduriama su dideliais duomenų kiekiais ir tokiomis problemomis kaip: sistemų integracija, kokybės kontrolė, duomenų standartizavimas, indeksavimas, saugojimas, perdavimas, pavaizdavimas ir t.t., bet dažnai nėra aišku kada kokią architektūrą ar technologiją pasirinkti, kokios jos yra, ar kuo jos skiriasi?

1.1. Išskirstytos architektūros

1.1.1. Modulinis programavimas

Tinkamas kompleksiškumo valdymas yra būtina dalis kuriant visas sistemas, išskyrus nereikšmingas taikomąsias programas. Viena iš fundamentaliausių technikų tam kompleksiškumui suvaldyti yra kodo organizavimas į susijusius funkcionalumo vienetus. Šią techniką galima taikyti daugelyje lygmenų, t.y. organizuoti kodą į procedūras; procedūras į klases; klases į komponentus; ir komponentus į didesnes susijusias posistemes. Išskirstytos sistemos iš to turi labai daug naudos ir net daugeliu atvejų padeda pasiekti šią koncepciją, nes modulumas yra būtinas norint išskirstyti kodą įvairioms mašinoms [GJT94]. Iš tikrųjų, dauguma išskirstytų architektūrų kategorijų daugiausia skiriasi tik pareigomis, paskirtomis skirtingiems moduliams ir jų sąveikoms.

1.1.2. Klientas-Serveris

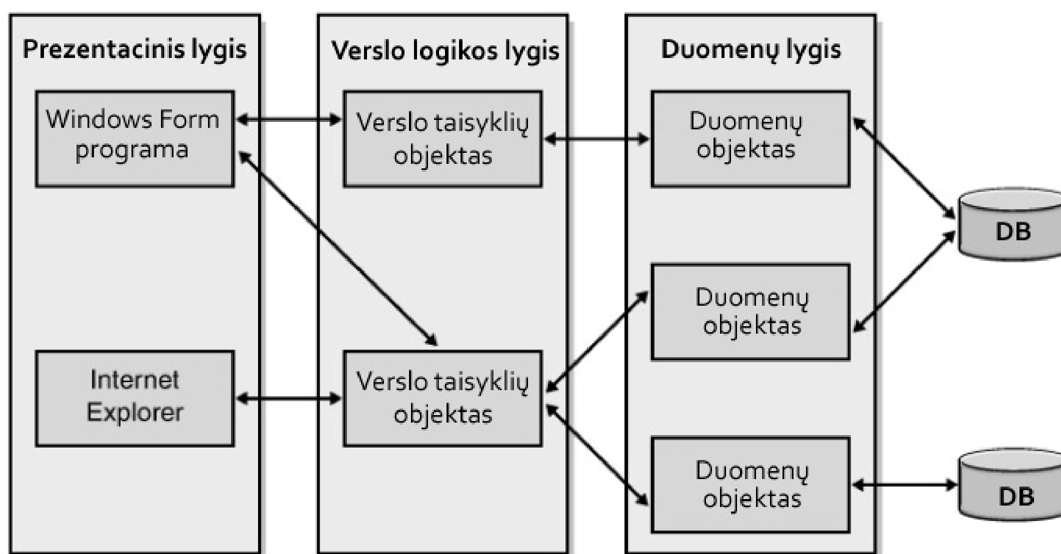
Klientas-serveris yra ankstyviausia ir svarbiausia išskirstyta architektūra. Bendrais terminais kalbant, klientas-serveris yra tiesiog kliento procesas, kuris prašo paslaugų iš serverio proceso. Kliento procesas tipiškai yra atsakingas už pristatymo sluoksnį (kitaip naudotojo interfeisą). Šis sluoksnis apima naudotojų įvesčių patikrinimą, kreipimųsi į serverį išsiuntimą, ir galbūt kai kurių verslo taisyklių įvykdymą. Serveris tada veikia kaip variklis, išpildantis kliento prašymus vykdydamas verslo taisykles ir sąveikaudamas su resursais tokiais kaip duomenų bazė ar failinė sistema. Dažniausiai su vienu serveriu bendrauja daug klientų [Ste02]. Nors mes nagrinėjame išskirstytas sistemas, reikia pastebėti, kad kliento ir serverio pareigos nebūtinai turi būti padalintos tarp kelių įrenginių. Taikomosios programos funkcionalumo atskyrimas yra taip pat geras projektavimo būdas ir procesams, veikiantiems ant vienos mašinos.

1.1.3. N Lygių (N-Tier)

Kliento-serverio programos taip pat dar vadinamos dviejų lygių programomis todėl, kad klientas bendrauja tiesiogiai su serveriu. Dviejų lygių architektūras yra paprastai gana lengva įgyvendinti, bet jas naudojant dažnai susiduriama su plečiamumo apribojimais. Praeityje, kūrėjai

apie N lygių architektūros reikalingumą dažnai suprasdavo iš tokio scenarijaus: programa veikė viename kompiuteryje ir kažkas nusprendė, kad dėl kažkokių priežasčių ją reikia išskirstyti. Šitos priežastys galbūt apėmė ketinimus aptarnauti daugiau kaip vieną klientą, suteikti prieigą prie bendrų resursų, ar panaudoti kiekvieno kompiuterio veikimo galią. Pradinės pastangos buvo pagrįstos dviejų lygių projektu — prototipas dirbo puikiai, ir buvo nuspręsta, kad viskas veikia gerai. Kadangi daugiau klientų buvo pridėta, veikimas pradėjo truputį sulėtėti. Dar didesnio kiekio klientų pridėjimas sistemą tiesiog parklupdė. Paskui, stengiantis išspręsti problemą, serverio aparatinė įranga buvo patobulinta, bet tai buvo brangus būdas ir tiktai užlaikė susidūrimą su tikrąja problema.

Šios problemos galimas sprendimas yra pakeisti architektūrą, kad ji panaudotų trijų lygių ar n lygių struktūrą. 1 pav. parodo, kaip trijų lygių architektūros apima vidurinio lygio, kuris turi vykdyti įvairias užduotis, pridėjimą į sistemą. Vienas iš pasirinkimų yra į vidurinį lygį padėti verslo logiką. Šiuo atveju, vidurinis lygis atlieka klientų pateiktų duomenų vientisumo tikrinimą ir jų apdorojimą remiantis verslo logika. Šis lygis gali apimti bendradarbiavimą su duomenų sluoksniu ar atmintyje vykdomų skaičiavimų vykdymą. Jei viskas suveikia gerai, vidurinis lygis paprastai pateikia savo rezultatus duomenų lygiui (laikymui) arba grąžina rezultatus klientui. Pagrindinė šios architektūros jėga yra aiškus vykdymo atsakomybių pasidalinimas.



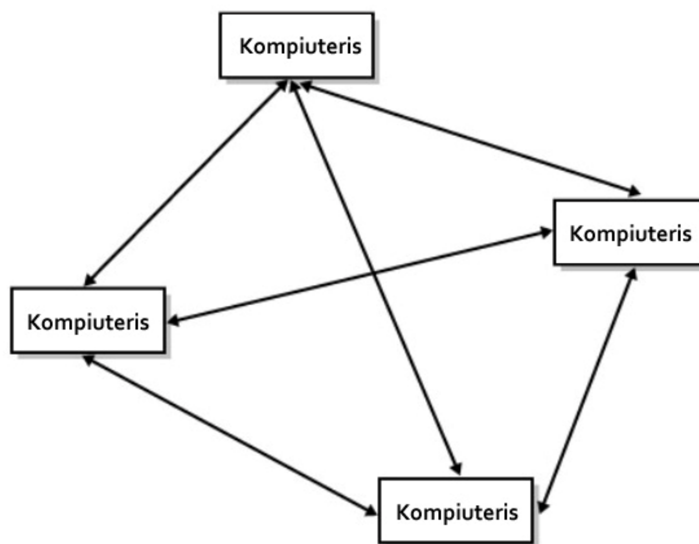
1 pav. Trijų lygių architektūros pavyzdys [PRP06]

Net jei daugiau kaip vienas n lygių sistemos lygis yra tame pačiame kompiuteryje, loginis sistemos funkcijų atskyrimas gali būti naudingas. Kūrėjai ar administratoriai gali palaikyti lygius atskirai, apkeisti juos ar migruoti juos į kitus kompiuterius, kad pritaikytų sistemos plečiamumą būsimiems poreikiams. Štai kodėl trijų lygių (ar iš tikrųjų n lygių) architektūros yra optimalios programinės įrangos palaikymui, išdėstymo ar dydžio keičiamumui, taip pat kaip ir lankstumui.

1.1.4. Lygus su lygiu (Peer-to-Peer)

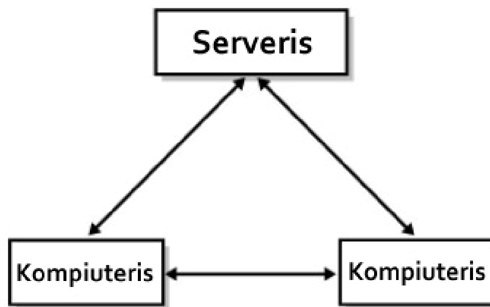
Prieš tai buvę išskirstytos architektūros turi aiškius vaidmenis kiekvienam iš lygių. Kliento-serverio lygiai gali būti lengvai pažymėti kaip šeimininkas-tarnas ar kaip gamintojas-naudotojas. Lygiai, n lygių modelyje, dažniausiai yra paskirstomi į vaidmenis tokius kaip pristatymo sluoksnis, verslo sluoksnis, ar duomenų sluoksnis, nors taip būti visada neprivalo. Kai kurie projektai gauna daugiau naudos iš labiau bendro modelio, kuriame aiškus suskirstymas vaidmenimis tarsi išblukęs, o kiekvienas mazgas yra lygiavertis. Scenarijai, tokie kaip darbo grupės, yra sukonstruoti būtent tokiu būdu, nes svarbiausia tokių išskirstytų sistemų funkcija yra pasidalinti informacija [GJT94].

Gryna lygus su lygiu (arba lygiarangių) architektūra susideda iš daugelio atskirų mazgų be centralizuoto serverio, kaip parodyta 2 pav., o visi susijungę naudotojai yra lygiaverčiai, kiekvienas veikia ir kaip klientas ir kaip serveris.. Neturint įprasto pagrindinio serverio, turi būti mechanizmas, kuris įgalina tuos įrenginius surasti vienam kitą. Tai paprastai pasiekiama naudojant transliavimo techniką ar kai kuriuos iš anksto apibrėžtus konfigūracijos nustatymus.



2 pav. Lygus su lygiu architektūra [GJT94].

Internetą paprastai laiko klasikine kliento-serverio architektūra su monolitiniu Web serveriu, aptarnaujančiu daug klientų, bet internetu veikia ir nemažai kvazi lygus su lygiu (quasi–peer–to–peer) programų, kuriomis šiandien naudojasi labai daug žmonių. Šitos sistemos leidžia bendrą duomenų pasidalinimą tarp dalyvaujančių kompiuterių. Jos naudoja centralizuotą serverį dalyvaujančių kompiuterių suradimui ir informacijos paieškai, kaip parodyta 3 pav.. Nors tai nėra gryna lygus su lygiu architektūra, šitie hibridiniai modeliai paprastai veikia daug geriau negu visiškai decentralizuotas lygus su lygiu modelis ir teikia tą pačią bendrą naudą.



3 pav. Kvazi lygus su lygiu architektūra [GJT94].

Pasirinkta architektūra yra tarsi gairė (būda) nusakanti kaip galima logiškai išskirstyti ir padalinti sistemos dalių vaidmenis, tačiau norint įgyvendinti vienokią ar kitokią architektūrą atitinkančią sistemą, reikia turėti tam skirtas priemones, todėl kitame skyriuje išnagrinėsime išskirstytų sistemų kūrimui skirtas technologijas.

1.2. Išskirstytos technologijos

Įvairios išskirstytos architektūros, kurias mes aptarėme, per ilgus metus buvo įgyvendinamos naudojant įvairias technologijas. Nors šios architektūros yra išbandytos ir teisingos, didelė pažanga išskirstytos programinės įrangos išsivystyme priklausė naujoms technologijoms. Palyginti su įrankiais ir abstrakcijomis, naudotomis, išskirstytų sistemų kūrimui prieš 10 metų, šiandieninės technologijos jas stipriai pralenkė. Šiandien galima praleisti daug daugiau laiko sprendžiant verslo logikos problemas negu reikia infrastruktūros konstravimui, tam, kad persiųstume duomenis iš vieno įrenginio į kitą.

Išskirstytų taikomųjų programų technologijos DCOM, Java RMI ar CORBA buvo plėtojamos daugelį metų, kad neatsiliktų nuo pastoviai didėjančių verslo reikalavimų. Šiandienėje aplinkoje, išskirstytų taikomųjų programų technologija turi būti efektyvi, praplečiama, palaikyti transakcijas, galėti operuoti su skirtingomis technologijomis, būti lengvai konfigūruojama, veikti per internetą ir daugiau. Bet ne visos sistemos yra tokios didėlės apimties, kad viso to palaikymo prireiktų. Kad palaikytų mažesnes sistemas, išskirstytos technologijos turi palaikyti bendrą išskirstytoms sistemoms būdingą elgesį ir būti paprastai konfigūruojamos, kad šitų sistemų išdalijimas būtų kiek galima lengvesnis. Atrodo, kad turėtų būti neįmanoma kažkuriai vienai technologijai patenkinti šį visą reikalavimų sąrašą. Iš tikrųjų dauguma šiandienių išskirstytų technologijų prasidėjo nuo žymiai kuklesnio reikalavimų sąrašo ir paskui per daugelį metų įgavo palaikymą kitiems reikalavimams. Išnagrinėkime kaip viskas išsivystė.

1.2.1. Prievadai (Sockets)

Prievadai (sockets) yra viena iš bazinių šiuolaikinių tinklo programų abstrakcijų. Prievadai apsaugo programuotojus nuo žemo lygio tinklo detalių, naudojant juos duomenų perdavimas atrodo kaip srautu (stream) pagrįsta įvestis ir išvestis. Nors prievadai suteikia absoliučią komunikacijos kontrolę, jie reikalauja per daug darbo, kuriant sudėtingas, didelių galimybių išskirstytas sistemas. Duomenų perdavimui naudojant srautu pagrįstą įvestį ir išvestį, kūrėjai turi konstruoti žinučių kūrimo ir perdavimo, bei įeities srautų interpretavimo sistemas. Toks darbas yra per daug varginantis daugumai nespecializuotų išskirstytų sistemų. Ko kūrėjams reikia tai yra aukštesnio lygmens abstrakcijos — tokios, kuri leidžia susidaryti įspūdį jog kviečiama vietinė funkcija ar procedūra.

1.2.2. Nutolusių procedūrų kvietimai

Open Group (anksčiau žinoma kaip Open Software Foundation) savo sukurtoje išskirstytų skaičiavimų aplinkoje (Distributed Computing Environment (DCE)), be kitų technologijų, apibrėžė ir nutolusių procedūrų kvietimų (Remote Procedure Calls (RPC)) specifikaciją. Su RPC, derama konfigūracija ir kai kuriais duomenų tipų apribojimais, kūrėjai galėjo įgyvendinti nutolusių procedūrų kvietimą naudojant daugelį tos pačios semantikos, reikalingos vietiniam procedūros iškvietimui. RPC įvedė kelias fundamentalias koncepcijas, kurios yra pagrindas visoms šiuolaikinėms išskirstytoms technologijoms, įskaitant DCOM, CORBA, Java RMI, ir .NET Remoting. Štai kai kurios iš tų pagrindinių koncepcijų:

- **Įgaliotiniai** (Stubs). Tai kodo dalys, kurios veikia kliento kompiuteryje ir per jas serverio nutolusių procedūrų kvietimas atrodo lyg yra vietinis. Pavyzdžiui, kliento kodas kviečia procedūrą iš į*galiotinių*, kuri atrodo lygiai taip pat kaip ir įgyvendinta serveryje, o įgaliotinis tada pasirūpina iškvietimo perdavimu pirmyn į nutolusią procedūrą.
- **Suskirstymas** (Marshalling). Tai yra parametrų perdavimo iš vieno konteksto į kitą procesas. Naudojant nutolusių procedūrų kvietimus, funkcijų parametrai yra suskirstomi serijomis į paketus taip juos paruošiant perdavimui per tinklą.
- **Interfeiso apibrėžimo kalba** (Interface Definition Language (IDL)). Ši kalba suteikia standartinę priemonę, leidžiančią apibrėžti nutolusių procedūrų kvietimo sintaksę ir duomenų tipus, nepriklausomai nuo kokios nors specifinės programavimo kalbos. IDL nėra būtinas pavyzdžiui Javai RMI todėl, kad ši išskirstyta technologija palaiko tiktai vieną kalbą — Java.

RPC reiškė didžiulį šuolį pirmyn darant nutolusią komunikaciją, draugiškesnę negu prievadų programavimas. Tačiau ilgainiui, programavimo pramonė nutolo nuo procedūrinio

programavimo iki objektiškai orientuoto kūrimo, normalu, kad atsirado ir tam skirtos išskirstytos objektinės technologijos.

1.2.3. Išskirstyti objektai

Šiandien dauguma kūrėjų pripažįsta objektiškai orientuotą programavimą kaip pagrindinį šiuolaikinio programinės įrangos vystymo principą. Kada tik žmonės susiduria su kažkuo sudėtingo kaip didelių programinės įrangos sistemų kūrimas, efektyvių abstrakcijų naudojimas yra kritiškas. Objektai yra pamatinės šiandien naudojamos abstrakcijos. Todėl, jei kūrėjai pripažįsta objektų naudą, tai reiškia kad jų naudojimas išskirstytiems scenarijams irgi ne išimtis.

Išskirstytos objektinės technologijos leidžia objektams, veikiantiems tam tikrame kompiuteryje būti pasiekiamiems kitų programų ar objektų, veikiančių kituose kompiuteriuose. Taip kaip RPC nutolusias procedūras leidžia naudoti kaip lokalias, taip ir išskirstytų objektų technologijos leidžia nutolusius objektus naudoti kaip lokalius. DCOM, CORBA, Java RMI, ir .NET Remoting yra išskirstyto objektinio programavimo technologijų pavyzdžiai. Nors pats šių technologijų įgyvendinimas ganėtinai skiriasi ir yra pagrįstas skirtinga verslo filosofija, daugeliu atvejų jos yra nepaprastai panašios:

- Jos yra pagrįstos objektais, kurie arba turi arba gali turėti būseną. Kūrėjai gali panaudoti nutolusius objektus su faktiškai ta pačia semantika kaip vietinius objektus. Tai supaprastina išskirstytą programavimą, suteikiant paprastą, vieningą programavimo modelį. Kur įmanoma, kūrėjai gali nepaisyti specifinių išskirstytam programavimui kalbos artefaktų ir perkelti juos į konfigūracijos sluoksnį.
- Jos yra sujungtos komponentiniu modeliu. Terminas komponentas gali būti apibrėžtas keliais būdais, bet čia komponentu vadinsime atskirą, dislokuojamą funkcionalumo vienetą. Komponentai atstovauja objektiškai orientuotų praktikų vystymąsi nuo baltosios dėžės pakartotinio naudojimo iki juodosios dėžės pakartotinio panaudojimo. Dėl jų griežtų viešų kontraktų, komponentai paprastai turi mažiau priklausomybių ir gali būti surinkti ir perkelti kaip funkciniai vienetai. Komponentų naudojimas padidina išdėstymo lankstumą taip pat kaip ir bendrų paslaugų tiekimo galimybes.
- Jos jungiasi su „didžiosiomis“ paslaugomis. „Didžiosios“ paslaugos tipiškai apima transakcijų, objektų kaupimo (pooling), lygiagretumo valdymo, ir objektų lokalizavimo palaikymą. Šitos paslaugos sprendžia bendrus didelių mastų sistemų reikalavimus ir jas yra gana sunku įgyvendinti. Kai klientinė dalis išskirstytoje sistemoje pasiekia tokį tašką šitos paslaugos tampa kritiškos plečiamumui ir duomenų vientisumui. Kadangi šias paslaugas įgyvendinti yra sudėtinga, o jos dažnai yra

būtinos, tai jos būna pateikiamos kartu su išskirstyta technologija ar operacine sistema ir programuotojams patiems nereikia jomis rūpintis.

1.2.4. DCE/RPC

Išskirstyta skaičiavimo aplinka (DCE), buvo sukurta ankstyvaisiais 1990-aisiais, kaip įrankių ir paslaugų rinkinys, kuris leido lengvesnį išskirstytų sistemų vystymą ir administravimą. DCE karkasas teikia kelias pagrindines paslaugas tokias kaip jau truputį aptarti nutolusių procedūrų kvietimai (DCE/RPC), saugumo bei laiko valdymas ir taip toliau.

DCE įgyvendinimas yra ganėtinai gąsdinanti užduotis: interfeisai turi būti apibrėžti interfeiso apibrėžimo kalba ir sukompiliuoti į C kalbos antraštes ir serverio įgaliotinius (stubs) naudojant IDL kompiliatorių. Realizuojant serverį norint susijungti su dvejetaine biblioteka būtina naudoti DCE gijas (DCE/Threads), kurios yra prieinamos tik C arba C ++ kalbose. Kitokių programavimo kalbų naudojimas yra apribotas dėl priklausomybės nuo pagrindinių paslaugų, tokių kaip DCE/Threads, todėl galų gale visi turi apsieiti su vienos gijos serveriais, jei nusprendžia nenaudoti C arba C++.

DCE/RPC vis dėlto yra pamatas daugeliui šiuolaikinių aukštesnio lygmens protokolų, įskaitant DCOM ir COM+. Keletas taikomųjų programų lygmens protokolų tokių kaip MS SQL Server, Server Message Block (SMB), naudojamas failų ir spausdintuvo pasidalinimui, ir Network File System (NFS) yra taip pat pagrįsti DCE/RPC.

1.2.4. CORBA

Objekto vadybos grupės (Object Management Grouop (OMG)), tarptautinio apytiksliai 800 kompanijų apimančio konsorciumo, suprojektuotas CORBA, kuris reiškia bendrą objekto prašymo brokerio architektūrą (Common Object Request Broker Architecture), yra tiktai standartų kolekcija, o objektų prašymų brokerių (ORB) realizacija tenka įvairioms trečiosioms šalims [Zah00]. Kadangi standarto dalys yra laisvai pasirenkamos, ir ORB kūrėjams leidžiama įtraukti papildomas ypatybes, kurių nėra specifikacijoje, tai viskas pasibaigė keliais nesuderinamais prašymų brokeriais. Rezultate, programa sukurta panaudojant vieno gamintojo ypatybes, negalėjo būti lengvai perkelta į kitą ORB. Jei perkama CORBA pagrįsta programa ar komponentas, negalima būti įsitikinusiems, kad jis veiks sujungs jį su esamomis CORBA programomis, kurios galbūt buvo sukurtos skirtingam ORB.

Nežiūrint į paminėtą problemą, CORBA taip pat turi ganėtinai sudėtingą studijavimo kreivę. Standartas skaitosi kaip norų sąrašas kuriame pilna visko, kas tik įmanoma naudojant nutolusius komponentus - kartais to yra tiesiog per daug "standartiniam verslui". Greičiausiai

bus praleista daug dienų ar savaičių standartų skaitymui prieš tai kol pirmas kvietimas bus nusiųstas į serverį.

Vis dėlto, sugebėjus realizuoti savo pirmą CORBA kvietimą, galima sujungti daug programavimo kalbų ir platformų. Yra net sluoksnių skirtų COM ar EJB integracijai, ir nepaisant SOAP, CORBA yra vienintelė tikra daugiaplatforminė, daugiakalbė aplinka išskirstytoms sistemoms.

1.2.4. DCOM

Išskirstytas komponentinis objektų modelis (Distributed Component Object Model (DCOM)) yra praplėtimas, telpantis į Komponentinio Objektų Modelio (COM) architektūrą, kuri yra dvejetainio sąveikavimo standartas, leidžiantis komponentais orientuotų išskirstytų sistemų kūrimą. Dažniausiai su COM susiduriama, naudojant ActiveX valdymo elementus ar ActiveX bibliotekas [Tha99].

DCOM leidžia tokių komponentų dalijimą tarp skirtingų kompiuterių. Plečiamumas, suvaldomumas, ir naudojimas dideliuose tinkluose iškelia kelias problemas, į kurias reikia atkreipti dėmesį. DCOM naudoja skimbtelėjimo (pinging) procesą, objekto egzistavimo trukmės suvaldymui t.y. visi klientai, kurie naudoja tam tikrą objektą siunčia atitinkamas žinutes kas tam tikrą intervalą. Kai serveris gauna šitas žinutes, jis žino, kad klientas yra vis dar veikiantis, kitaip jis sunaikintų nebenaudojamą objektą.

1.2.5. MTS/COM+

COM+ technologija, anksčiau žinoma kaip Microsoft Transaction Server (MTS), buvo pirmosios rimtos Microsoft kompanijos pastangos įsiterpti į didelių, įmonės lygio sistemų kūrimo sritį. Ji tarnauja ne tik kaip išskirstyta platforma, bet taip pat aprūpina transakcijų, saugumo, plečiamumo ir išdėstymo servisais. COM+ komponentai gali būti panaudoti net per Microsoft Message Queue Serverį, kas leidžia asinchronišką metodų vykdymą.

Nepaisant pranašumų, COM+ nepalaiko automatinio objektų apdorojimo, kad būtų galima juos perduoti tarp programų pagal jų reikšmę, vietoj to savo duomenų struktūrą reikia perduoti, naudojant ADO recordsets ar kitas serializavimo priemones. Kiti trūkumai, kurie laiko žmones toliau nuo COM+ naudojimo yra šiek tiek sudėtinga konfigūracija ir išdėstymas, kuris komplikuoja COM+ naudojimą realioms sistemoms.

1.2.6. Java RMI

Tradicinis Java nutolusių metodų kvietimas (Remote Method Invocation (Java RMI)) naudoja rankų darbo proxy/stub kompiliacijos ciklą. Skirtingai nuo DCE/RPC ir DCOM,

interfeisai yra aprašyti ne abstrakčia IDL kalba, bet naudojant Java. Tai įmanoma dėl to, kad Java yra vienintelė kalba, kuria galima rašyti RMI realizacijas [Gro01].

Šis apribojimas išstūmė RMI iš didelių išskirstytų sistemų integracijos žaidimo. Net nepaisant to, kad visos aktualios platformos palaiko Javos Virtualią Mašiną, integracija su jau sukurtomis sistemomis nėra lengvai įgyvendinama.

1.2.7. Java EJB

Enterprise Java Beans (EJB) buvo Sun korporacijos atsakymas į Microsoft COM+. Skirtingai nuo CORBA, kadangi tai yra tiktai standartas, EJB yra su įgyvendinta nurodymų realizacija. Tai leidžia kūrėjams patikrinti, ar jų produktai veikia kuriame nors standartą atitinkančiame EJB konteineryje. EJB buvo plačiai priimtas pramonėje, ir yra net keletas konteinerių realizacijų, pradedant nuo nemokamų atviro kodo projektų iki komercinių, kuriuos pateikia žinomi tarpinės programinės įrangos gamintojai [RBS05].

Viena problema su EJB yra ta, kad net nepaisant egzistuojančios standartus atitinkančios realizacijos, dauguma gamintojų prideda savo ypatybes prie taikomųjų programų serverių. Jei kūrėjas parašo komponentą, kuris naudoja vieną iš tų ypatybių, visa programa automatiškai jau nebeveiks naudojant kito gamintojo EJB konteinerį.

Ankstesnės EJB versijos buvo apribotos tik Java platformoje dėl jų vidaus persipynimo su RMI. Dabartinė versija leidžia naudoti IIOP, tai yra tas pats perdavimo protokolas kurį naudoja CORBA, be to ir trečiosios šalys pateikia komercinius COM/EJB tiltus.

1.2.8. Web Servisai/SOAP/XML-RPC

Web servisai buvo pirma lengvai suprantama ir įgyvendinama platforma, skirta kurti tikrai nepriklausančioms nuo platformos ir kalbos sistemoms. Web servisai techniškai yra būsenos neturintys nutolusių komponentų kvietimai per HTTP POST su duomenimis, užkoduotais kokiame nors XML formate.

Šiuo metu daugiausiai naudojami du skirtingi XML kodavimai: XML-RPC ir SOAP [STK02]. XML-RPC gali būti apibūdintas kaip neturtingo žmogaus SAOP. Jis apibrėžia labai lengvą ir paprastą protokolą, kurio specifikacija užima tik maždaug penkis lapus. Įvairios realizacijos jau egzistuoja ir yra skirtos daugeliui programavimo aplinkų, tokių kaip AppleScript į C/C ++, COM, Java, Perl, PHP, Python, Tcl, Zope ir .NET.

SOAP, arba kitaip paprastas objekto prieigos protokolas (Simple Object Access Protocol), apibrėžia daug platesnį paslaugų komplektą. Specifikacija apima ne tiktai nutolusių procedūrų iškvietimus, bet taip pat ir Web servisų apibrėžimų kalbą (Web Service Description Language (WSDL)) ir visuotinį apibūdinimą, atradimą, ir integraciją (Universal Description, Discovery and

Integration (UDDI)). WSDL yra SOAP interfeiso apibrėžimo kalba, o UDDI veikia kaip direktyvinė paslauga web servisų atradimui. Šie papildomi protokolai ir specifikacijos yra taip pat pagrįsti XML, kuris leidžia visoms SOAP ypatybėms būti įgyvendintoms daugelyje platformų.

Specifikacijos ir straipsniai skirti SOAP, WSDL, UDDI ir kitoms atitinkamoms technologijoms sudaro kelis šimtus puslapių, ir galima tikėtis, kad šis skaičius ir toliau augs, kai bus nagrinėjamos tokios temos kaip maršrutizavimas ir transakcijos.

1.2.9. .Net Remoting

Iš pirmo žvilgsnio .Net Remoting santykis su web servisais, yra toks pat koks buvo tarp ASP ir CGI programavimo. Jis, palyginus su web servisais, išsprendžia daug svarbių problemų už kūrėjus, pavyzdžiui .Net Remoting technologija įgalina būseną turinčių objektų naudojimą. Vien šis faktas leidžia jai būti tvirtu šiuolaikinių ir ateities paskirstytų sistemų pagrindu [Bar02].

Be to, kad leidžia valdyti būseną turinčius objektus .Net Remoting turi lankstų ir išplečiamą karkasą, kuris leidžia naudoti skirtingus perdavimo mechanizmus (HTTP, ir TCP yra palaikomi pagal nutylėjimą), kodavimus (SOAP ir dvejetainis realizuoti iš karto), ir saugumo nustatymus (IIS Saugumas, ir SSL) [Mac03].

Su šiais pasirinkimas, ir jų visų išplėtimo ar visiškai naujos realizacijos sukūrimo galimybe .Net Remoting labai tinka šiandienėms išskirstytoms sistemoms. Galima lengvai pasirinkti tarp HTTP/SOAP internetui ar TCP/dvejetainio lokaliam tinklui, pakeičiant tik vieną eilutę konfigūracijos faile.

2. Technologijų principų analizė

Aišku, kad idealiausia būtų išnagrinėti ir ištirti visas egzistuojančias išskirstytų sistemų technologijas, nustatyti, kaip jos veikia, kokie yra jų pranašumai, veikimo greitis ir t.t., tačiau tai būtų labai didelės apimties darbas, be to dalis technologijų jau yra pasenusios ir retai kur benaudojamos, todėl šiame darbe, kaip tyrimo objektai, buvo pasirinktos priimtiniausios, lygiavertės ir šiuo metu vienos iš populiariausių išskirstytų sistemų technologijų – tai Java RMI ir .NET Remoting. Taigi visas tolimesnis darbas bus susijęs tik su šių dviejų technologijų analizavimu.

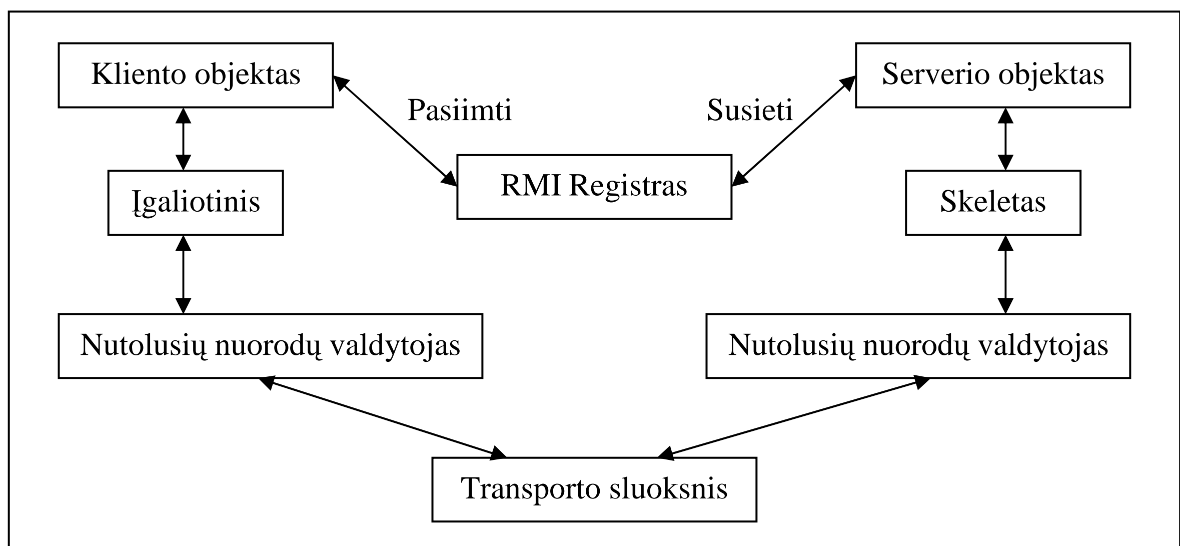
2.1. Java RMI

Java RMI leidžia mums iškviesti objekto, egzistuojančio kitoje adresų erdvėje, metodą, kuris gali būti patalpintas tame pačiame ar skirtingame kompiuteryje. Paprastai RMI programa

dažniausiai būna sudaryta iš dviejų atskirų programų: serverio ir kliento. Tipinė serverio programa sukuria nutolusius objektus, nuorodas į juos padaro prieinamas nutolusiems klientams ir laukia kol klientinės programos iškvies šiuos nutolusių objektų metodus. Tuo tarpu tipiška klientinė programa gauna nuorodą į vieną ar daugiau nutolusių objektų serveryje ir tada kviečia jų suteikiamus metodus [Gro01]. RMI šioje vietoje suteikia visą mechanizmą reikalingą nutolusiai komunikacijai tarp programų parašytų Java programavimo kalba.

Tam, kad klientas galėtų pirmą kartą surasti nutolusius serverio objektus, RMI naudoja centrinį vardą arba adresų katalogo, kuris laiko informaciją apie prieinamus objektus, paslaugas [RBS05]. Serverio objektas padaro savo metodus prieinamus nutolusiems kvietimams, susiedamas juos su vardais vardų saugykloje, o klientas gauna objektų nuorodas į Java/RMI serverį pasiimdamas jas iš vardų saugyklos. Gavęs nutolusio objekto nuorodą klientas kviečia serverio objekto metodus taip tarsi jie priklausyto jo pačio adresų sričiai.

RMI priklauso nuo dviejų panašių objektų tipų, kuriuos RMI kompiliatorius automatiškai generuoja iš serverio realizacijos: skeletai ir įgaliotiniai. Įgaliotinis – tai kliento pusės objektas kuris veikia kaip tiltas į nutolusį serverio objektą ir turi nutolusio objekto interfeisą. Jis atsakingas už duomenų surikiavimą ir padalinimą kliento pusėje. Skeletas - tai serverio pusės objektas persiuntinėja visus kvietimus tikrajai nutolusio objekto realizacijai ir yra atsakingas už duomenų surikiavimą ir padalinimą serverio pusėje. 1 pav. galime matyti pagrindinę RMI mechanizmo architektūrą ir komponentų išsidėstymą joje.



4 pav. Java RMI architektūra ir veikimo schema

2.1.1. Paprastos išskirstytos sistemos pavyzdys naudojant Java RMI

RMI programos kūrimas paprastai apima šiuos žingsnius:

1. „Serverio“ interfeiso, kuris gali būti naudojamas serverio objekto prieigai už vietinės Java virtualios mašinos ribų, projektavimas ir kūrimas

2. Išeities kodo kompiliavimas ir įgaliotinio, serverio kvietimui bei skeleto kvietimų priėmimui ir perdavimui, generavimas;
3. Serverio objektų padarymas prieinamais per tinklą naudojant RMI registrą arba JNDI (Java Naming and Directory Interface) vardų interfeisą.;
4. „Kliento“ kodo, kuris prisijungia prie serverio objekto ir jį iškviečia, sukūrimas.

Žemiau panagrinėsime paprastą programą, kad lengviau suprastumėme šiuos žingsnius.

Nutolęs interfeisas apibrėžia metodus kuriuos klientas gali kviesti iš kitų JVM. Šis interfeisas turi būti paveldėtas iš java.rmi.Remote interfeiso ir turi deklaruoti java.rmi.RemoteException prie galimų išskirtinių situacijų deklaracijų (throws), nes nutolusių metodų kvietimas gali nepavykti dėl su tinklu susijusių susisiekimo problemų ar serverio problemų.

```
public interface BankoValdymas extends Remote
{
    public BankoSaskaita getSaskaita(String saskaitosNr)
                                throws RemoteException, TransactionException;
    public void Inesti(BankoSaskaita iSask, float kiekis)
                                throws RemoteException, TransactionException;
    public void Isimti(BankoSaskaita isSask, float kiekis)
                                throws RemoteException, TransactionException;
}
```

Nutolusio interfeiso realizacija turi apibrėžti nutolusio objekto konstruktorių ir aprašyti visų metodų, kuriuos bus galima iškviesti iš kliento, realizacijas. Tam, kad sukurtumėme visą laiką veikiantį nutolusį objektą, kuris sąveikai naudotų RMI numatytąjį prievadais pagrįstą transportą, galime realizacijos klasę paveldėti iš java.rmi.server.UnicastRemoteObject klasės.

```
public class BankoValdymasReal extends UnicastRemoteObject
implements BankoValdymas
{…}
```

Serverio klasė sukuria nutolusio objekto egzempliorių ir jį susieja su vardu RMI registre arba JNDI.

```
public class BankoValdymasReg {
    public static void main(String[] args){
        try{
            BankoValdymasReal bankas = new BankoValdymasReal("Snoras");
            Naming.rebind("Snoras", bankas);
        } catch (…)
    }
}
```

Klientas iš RMI registro pasiima objekto nuorodą ir tada vykdo nutolusių metodų kvietimus lygiai taip kaip kviestų paprastus vietinius metodus.

```
public class BankClient{
    public static void main(String[] args) {
        try{
```

```
                BankoValdymas bankas = (BankoValdymas)Naming.lookup("Snoras");
                BankoSaskaita sask = bankas.getSaskaita("0007");
                …
                bankas.Inesti(sask,5000);
             }catch (…)
        }
    }
```

2.1.2. Duomenų perdavimas

Visi nutolę objektai yra perduodami pagal nuorodą, o objektai perduodami kaip metodų parametrai gali būti perduodami arba pagal nuorodą arba pagal reikšmę. Objektai kurie realizuoja Serializable interfeisą yra perduodami pagal reikšmę, o realizuojantys Remote interfeisą, pagal nuorodą. Neprimityvių tipų argumentai kurie nėra nei Remote nei Serializable negali būti perduodami. Pagrindinė procedūra kurią klientas naudoja sąveikaudamas su serveriu yra tokia:

1. Klientas susikuria įgaliotinio klasės egzempliorių. (kuris jau buvo prieš tai sugeneruotas ir turi visus metodus kuriuos ir serverio klasė)
2. Klientas kviečia įgaliotinio metodą.
3. Įgaliotinis arba sukuria naują arba panaudoja jau egzistuojantį prisijungimą per prievadą prie serveryje esančio skeleto. Jis surenka ir paruošia visą su metodo kvietimu susijusią informaciją, apimant metodo vardą ir argumentus, ir siunčia šią informaciją kaip baitų masyvą per prievado prisijungimą serverio skeletui.
4. Skeletas gautus duomenis interpretuoja, iš baitų atstato į reikiamą formą ir įvykdo tikrojo serverio objekto metodo iškvietimą. Jis atgal gauna grąžinamą reikšmę iš to pačio tikrojo objekto, ją paruošia perdavimui ir išsiunčia atgal metodą kvietusiam įgaliotiniui.
5. Įgaliotinis išanalizuoja duomenis, atstato grąžintą reikšmę į reikiamą formatą ir grąžina ją kliento kodui.

2.1.3. Laido protokolas, RMI-IIOP

JRMP (Java Remote Method Protocol) dar vadinamas laido protokolu (wire protocol) yra uždaras Sun protokolas, sukurtas kaip virtualus laidas panaudojant TCTP/IP protokolą ir yra skirtas serializuotų RMI duomenų persiuntimui. RMI per IIOP (Internet Inter-ORB Protocol) yra Java 2 platformos dalis ir šis protokolas įgalina sąveiką tarp Java RMI ir CORBA objektų. RMI-IIOP veikia panašiai kaip ir Java RMI išskyrus faktą, kad jis naudoja IIOP protokolą, kaip pamatą komunikacijai, ir ORB (Object Request Broker) vardų paslaugoms vietoj Java RMI registro.

2.1.4. Vardų paslaugos

RMI gali naudoti daug skirtingų vardų arba katalogų paslaugų įskaitant Java vardų ir katalogų interfeisą (JNDI). Pačiame RMI yra paprasta paslauga pavadinta RMI Registru – rmiregistry. RMI registras veikia kiekviename kompiuteryje, kuris viešina nutolusius objektus, ir priima užklausas dėl paslaugų, numatytuoju atveju veikia naudodamas 1099 jungtį (port). RMI registras paprastai yra paleidžiamas kaip atskirai veikiantis serveris. Panagrinėkime kaip visa tai veikia.

Tarkime turime veikiantį vardų registrą. Serverio kompiuteryje, serverio programa sukuria nutolusią paslaugą iš pradžių sukurdama lokalų objektą kuris realizuoja reikiamą interfeisą. Vėliau tas objektas yra eksportuojamas susiejant jį su vardu registre, ir tada objektas jau yra prieinamas klientų kvietimams. Kliento pusėje RMI registras yra pasiekiamas naudojant statinę klase *Naming*. Ši klasė turi metodą *lookup()*, kurį naudoją klientinė programa vardų registro užklausoms. *Lookup()* metodas priima universalų adresą (pvz.: rmi://<kompiuterio_vardas>[:jungtis]/paslaugos_vardas), kuris apibrėžia serverio vardą ir pageidaujamos paslaugos vardą, ir grąžina nuorodą į nutolusios paslaugos objektą.

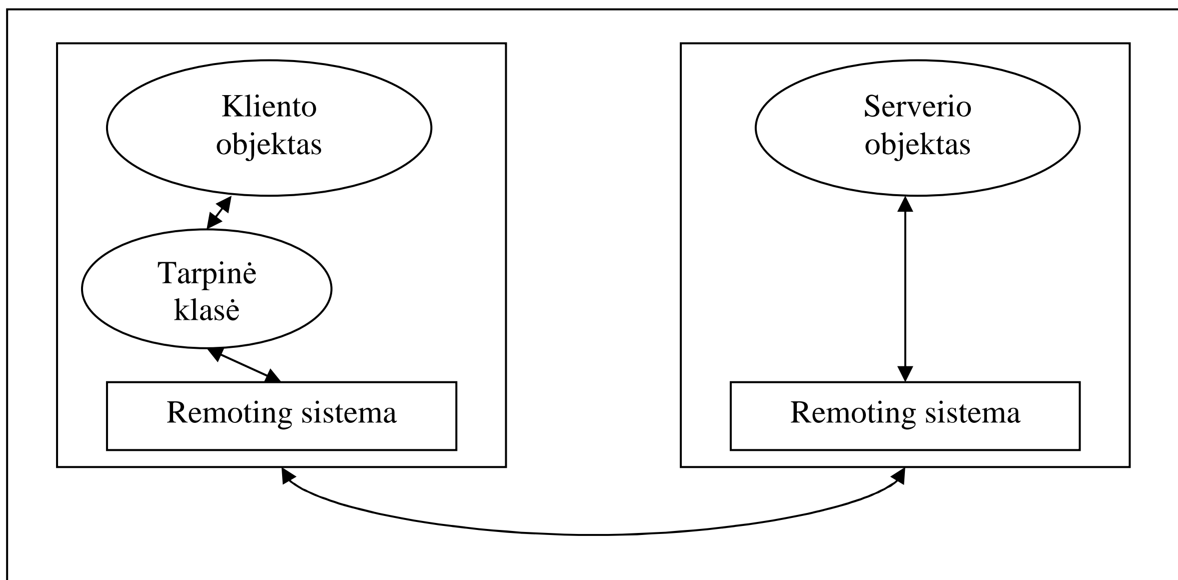
2.2. .Net Remoting

.Net Remoting yra visapusiška išskirstytų sistemų kūrimo technologija. Naudojantis šia technologija galima kurti programas, kurios naudoja objektus iš kitų adresų erdvių tame pačiame arba bet kuriame kitame per tinklą pasiekiamame kompiuteryje [Bar02]. .Net Remoting yra labai lengva pritaikyti, kad atitiktų esančius reikalavimus. Pavyzdžiui galima pasirinkti skirtingus komunikacijos protokolus (pvz.: HTTP, TCP), skirtingus serializavimo formatus (pvz.: SOAP formuotoją, dvejetainį formuotoją), skirtingus iškvietimo būdus (pvz.: sinchroninis, asinchroninis), skirtingus serverio objektų aktyvavimo būdus (pvz.: kliento aktyvuotas, serverio aktyvuotas), objektų gyvavimo ciklus ir t.t. [Mac03].

Norint sukurti programą, kurioje du komponentai sąveikauja tiesiogiai, nepaisant programos srities ribų, naudojant .Net Remoting, viskas ko reikia yra:

- Nutolusio objekto, kurio metodai gali būti kviečiami iš skirtingos adresų erdvės.
- Programos, kuri paleista lauktų to objekto užklausų.
- Kliento programos, kuri kviestų tą objektą.

Kaip ir Java RMI, .Net Remoting komunikacijai naudoja tarpines klases, tik šiuo atveju tam sukuriama tik viena klasė kuri yra kliento pusėje, kaip parodyta 2 pav.



5 pav. .Net Remoting architektūra ir veikimo schema

2.2.1. Paprastos išskirstytos sistemos pavyzdys naudojant .Net Remoting

Šiame skyriuje išnagrinėsime kaip sukurti banko valdymo programą aprašytą Java RMI tik dabar tam naudojant .Net Remoting.

Nutolęs objektas: ši klasė turi būti paveldėta iš *MarshalByRefObject* objekto, tai yra pagrindinė klasė objektams, kurie keičiasi žinutėmis naudodami įgaliotąją programą. Objektai kurie ne paveldi *MarshalByRefObject* besąlygiškai tvarkomi pagal reikšmę. Žemiau esančiame kodo pavyzdyje aprašytas paprastas objektas kuris gali būti sukurtas ir iškviestas objektų veikiančių kitoje programos srityje.

```
public class BankoValdymas : MarshalByRefObject
{
    public BankoValdymas()
    {…}
    public BankoSaskaita getSaskaita(string saskaitosNr)
    {…}
    public void Inesti(BankoSaskaita iSask, float kiekis)
    {…}
    public void Isimti(BankoSaskaita isSask, float kiekis)
    {…}
}
```

Serveris turi suderinti serverio .Net Remoting sistemą nustatant aktyvavimo būdą ir kitą informaciją, tokią kaip programos vardas ir prieigos taškas. Yra du būdai tam padaryti: viską nustatyti programiniu būdu, arba naudoti konfigūracinį failą. Taip pat serveriui reikia sukurti tinkamą kanalą ir užregistruoti jį sistemoje. Objektas gali būti naudojamas dviem budais: kaip vienas egzempliorius (singleton) ir kaip vieno kvietimo objektas (singlecall). Kai naudojamas vienas egzempliorius, objektas yra sukuriamas pirmo kliento kreipimosi metu ir išlieka iki kol klientas nutraukia susijungimą. Vieno kvietimo atveju objektas yra sukuriamas kiekvieno kliento metodo kreipimosi metu, ir išlieka tik iki kol metodas baigia darbą.

```
public class Serveris {
   public static void Main() {
        ChannelServices.RegisterChannel(new TcpChannel(1234));
        RemotingConfiguration.RegisterWellKnownServiceType(
           typeof(BankoValdymas),
           "Snoras",
           WellKnownObjectMode.Singleton);
        Console.WriteLine("Laukiama užklausų...");
        Console.ReadLine();
   }
}
```

Kaip alternatyvą galima naudoti konfigūracinį failą.

```
//Serveris.exe.config
<configuration>
   <system.runtime.remoting>
        <application>
             <service>
                  <wellknown
                       mode="Singleton"
                       type="BankoValdymas, BankoValdymas"
                       objectUri="Snoras.rem"/>
             </service>
             <channels>
                 <channel ref="http" port="1234"/>
             </channels>
        </application>
   </system.runtime.remoting>
</configuration>
```

Tada serverio *main* metode reiktų iškviesti tokią komandą.

```
RemotingConfiguration.Configure("Listener.exe.config");
```

Klientas turi nustatyti kliento .Net Remoting sistemos parametrus, pvz.: aktyvavimo būdą ir kitą tipų informaciją, tokią kaip programos vardą ir objekto URI (Uniform Resource Identifier). Taip pat klientas irgi turi sukurti atitinkamą kanalą ir jį užregistruoti sistemoje. Vėl gi yra du būdai tam padaryti – arba viską nustatyti programiškai arba naudoti konfigūracinį failą.

```
public class Klientas{
   public static void Main() {
      ChannelServices.RegisterChannel(new TcpChannel());
      RemotingConfiguration.RegisterWellKnownClientType(
         typeof(BankoValdymas),
         "tcp://localhost:1234/Snoras");
      try{
         BankoValdymas bankas = new BankoValdymas();
         BankoSaskaita sask = bankas.getSaskaita("0007");
         bank.Inesti(sask, 5000);
         bank.Isimti(sask, 10);
      }catch (Exception e){
          Console.WriteLine(e);
      }
   }
}
```

Vėlgi galime naudoti konfigūracinį failą

```
//Klientas.exe.config
<configuration>
     <system.runtime.remoting>
          <application>
               <client>
                   <wellknown
                       type="BankoValdymas, BankoValdymas"
                       url="tcp://localhost:1234/Snoras.rem"/>
               </client>
          </application>
     </system.runtime.remoting>
</configuration>
```

Tada kliento *main* metode reikia naudoti tokią komandą

```
RemotingConfiguration.Configure("Client.exe.config");
```

2.2.2. Duomenų perdavimas

Nutolę objektai gali būti perduodami klientams tik pagal nuorodą, bet objektai perduodami kaip metodų parametrai gali būti perduodami arba pagal nuorodą arba pagal reikšmę. Norint perduoti objektus pagal reikšmę, jie turi būti pažymėti *[Serializable]* atributu, arba turi realizuoti *ISerializable* interfeisą. Visas objektas yra serealizuojamas ir perduodamas iš vienos pusės į kitą, o tada objektas deserializuojamas. Pagal reikšmę perduoti objektai tampa vietiniais, ir visi kviečiami jo metodai veikia toje pačioje adresų erdvėje.

Norint perduoti objektą pagal nuorodą, reikia, kad tas objektas būtų paveldėtas iš *MarshalByRefObject* klasės. Yra du tokių objektų tipai: serverio aktyvuojami objektai (SAO) ir kliento aktyvuojami objektai (CAO). SAO gali būti pažymėti kaip vieno egzemplioriaus ar vieno kvietimo, kaip jau minėta anksčiau. CAO atveju, kai klientas prašo sukurti objektą, naudodamas *Activator.CreateInstance()* arba operatorių *new* aktyvavimo žinutė yra nusiunčiama į serverį, kur nutolęs objektas ir sukuriamas.

2.2.3. HTTP kanalas ir TCP kanalas

Naudojant .Net Remoting technologiją objektai sąveikauja per taip vadinamus kanalus. Microsoft integravo kanalo koncepciją į Remoting siekiant išskirstymo modelį padaryti visiškai skeletinį. Visa tai skirta tam, kad būtų galima apibrėžti įvairius protokolus priklausomai nuo poreikių. Tai gali būti TCP/IP kanalas arba HTTP kanalas.

TCP kanalai yra greitesni ir rekomenduojami tinklams kuriuose ugniasienės turi nedaug apribojimų per tinklą keliaujantiems paketams. TCP kanalas naudoja dvejetainį formatuotoją žinučių serializavimui į dvejetainį srautą, o tą srautą į nurodyta adresą perduoda TCP protokolu. TCP kanalą taip pat įmanoma sukonfigūruoti taip, kad jis naudotų SOAP formatuotoją.

Iš kitos pusės HTTP kanalai yra daug lankstesni ir yra rekomenduojami aplinkoms kuriose nutolusi komunikacija vykdoma per internetą. HTTP kanalas perduoda žinutes į nutolusius objektus ir iš jų naudodamas SOAP protokolą. Visi duomenys būna apdorojami naudojant SOAP formatuotoją, kuris juos pakeičia į XML, serializuoja ir prideda reikalingas antraštes prie duomenų srauto. Taip pat yra įmanoma sukonfigūruoti HTTP kanalą, kad jis naudotų dvejetainį formatuotoją. Pilnai paruoštų duomenų srautas yra siunčiamas nurodytu adresu per HTTP protokolą.

3. Java RMI ir .Net Remoting veikimo principų palyginimas

Tiek Java RMI tiek .Net Remoting suteikia mechanizmus aiškiam nutolusių objektų, išskirstytų per tinklą, pasiekimui ir kvietimui. Nors naudojami vidiniai mechanizmai nutolusiai komunikacijai užtikrinti ir skiriasi, bet bendra idėja yra daugiau mažiau panaši. Žemiau esančioje 1 lentelėje pateikiama Java RMI ir .Net Remoting panašumų ir skirtumų, kuriuos detaliau išnagrinėsime šioje dalyje, santrauka.

1 lentelė. Pagrindinių Java RMI ir .Net Remoting veikimo principų santrauka

	Java RMI	**.Net Remoting**
Komunikacija	Prievadai (socket)	Kanalai (channel)
Konfigūracija	Sistemos parametrai	XML failas
Protokolai	JRMP, IIOP	HTTP, TCP, SOAP
Aktyvavimas	Activatable	Singleton, SingleCall, kliento aktyvuojamas
Formatas	Serializacija	SOAP arba dvejetainė serializacija
Išskirstytas šiukšlių surinkėjas	Taip	Taip
Klaidos	Kyla nutolusi nenumatyta situacija	Kyla nutolusi nenumatyta situacija

Kaip matome didžiausias dėmesys skiriamas būtent technologiniams aspektams, kurie leistų įvertinti abiejų technologijų pranašumus, trūkumus, sudėtingumą ir galimą veikimo greičio santykį. Nors, kaip jau minėta anksčiau, idėjiškai šios technologijos yra artimos ir panašios, bet iš pateiktų aspektų santraukos lentelės matome, kad sutampančių eilučių yra mažiau nei besiskiriančių, o tai reiškia kad skirtumų tarp šių technologijų yra daugiau nei panašumų. Taigi pirmiausiai apžvelgsime panašumus, o po to skirtumus.

3.1. Panašumai

3.1.1. Objektiškai orientuoti RPC

Nutolusių procedūrų kvietimai (RPC – Remote Procedure Calls) yra tradicinis mechanizmas leidžiantis programoms kviesti procedūras kituose kompiuteriuose. RPC naudojasi įgaliotiniais metodais, kurie turi tokį pat aprašą, kaip ir nutolę metodai, bet tuo pačiu turi ir kodą, skirtą duomenų persiuntimui tarp kliento ir serverio. Parametrai apdorojami ir siunčiami į serverį, kur jie atstatomi ir perduodami kviestam metodui. Su grąžinamomis reikšmėmis elgiamasi taip pat.

Galima pagalvoti, kad tiek Java RMI, tiek .Net Remoting yra tik objektiškai orientuoti būdai sukurti panaudojant jau egzistuojantį mechanizmą, nes RPC leidžia kviesti tik tinkle esančias procedūras, o RMI ir .Net Remoting leidžia kviesti objektų metodus. Objektai yra siuntinėjami pirmyn ir atgal tarp kliento ir serverio kaip parametrai ir grąžinamos reikšmės.

3.1.2. Interfeiso apibrėžimo kalba

Nei Java RMI nei .Net Remoting nereikalauja antrinės kalbos nutolusių objektų interfeisų apibrėžimui, tokios kaip CORBA IDL (Interface Definition Language). Kadangi Java RMI yra susieta su Java kalba, o .Net Remoting yra smarkiai susijęs su C# kalba, tai abi technologijos palaiko vienos klasės ir daugelio interfeisų paveldėjimą.

3.1.3. Serverio objektų gyvavimo ciklas

RMI yra realizuotas paskirstytas šiukšlių surinkėjas (Distributed Garbage Collector). Serveris stebi ir skaičiuoja klientus kurie turi veikiančius savo įgaliotinius. Jei klientas tyčia nutraukia ryšį tas skaičius yra sumažinamas vienetu. Jei aktyvių nuorodų skaičius į objektą pasiekia nulį - objektas pašalinamas. .Net Remoting taip pat turi išskirstytą šiukšlių surinkimą paremtą aktyvių nuorodų skaičiavimu.

3.2. Skirtumai

3.2.1. Nutolusių objektų nuorodų saugojimas

Java RMI nutolusių objektų nuorodų saugojimui naudoja katalogą, kuris vadinamas RMI registru. Šis registras, pats būdamas išskirstytu objektu, turi būti paleidžiamas dinamiškai arba rankiniu būdu prieš bet kokią sąveiką tarp kliento ir serverio.

.Net tam naudoją skirtingą būdą. Užuot naudojus vardų serverius, klientas susisiekia su serveriu naudodamas iš anksto nustatytą jungtį, todėl išskirstytos sistemos sukurtos naudojant .Net Remoting nepriklauso nuo jokiu trečių paslaugų serveriui surasti.

3.2.2. Nutolusių objektų deklaravimas ir realizavimas

Esminis skirtumas skiriantis abi technologijas yra tas, kad RMI reikalauja nutolusio objekto interfeiso apibrėžimo, kai tuo tarpu .Net Remoting nereikalauja, tačiau jis turi kitą trūkumą - kai jungiamasi prie nutolusių objektų .Net Remoting gali iškviesti tik numatytąjį to objekto konstruktorių (jei objektas turi kelis konstruktorius bus iškviestas pagrindinis). Java RMI tokio apribojimo neturi.

3.2.3. Programavimo lengvumas

Turbūt daug kas priklauso nuo asmeninės patirties. Greičiausiai programuojantiems Java parašyti RMI programą būtų lengviau, na o programuojantiems .Net technologijomis atvirkščiai. Kaip bebūtų, faktas yra tas, kad Naudojant Visual studio 2008 įrankį, paprastai .Net Remoting programėlei sukurti ir paleisti reikia mažiau nei pusvalandžio, o visas darbas kurį reikia atlikti – tai parašyti tinkamas klases, jas sukompiliuoti ir paleisti. Tačiau kuriant analogišką Java RMI programą reikia atlikti daugiau žingsnių norint paleisti klientą ir serverį. Tam dar papildomai reikia susitvarkyti su saugumo politika, RMI registru, įgaliotinių ir skeletų generavimu. Nors tai yra gana subjektyvu, tačiau žmogui neturinčiam patirties nei su viena iš technologijų, .Net Remoting programuoti turėtų būti lengviau nei Java RMI. Be to .Net Remoting suteikia labiau išbaigtą ir lankstesnį sprendimą išskirstytų sistemų kūrimui, nes galima labai lengvai pakeisti protokolą, serializavimo būdą ir panašiai.

3.2.4. Skirtingų kalbų ir platformų sąveika

.Net Remoting gali būti naudojamas tik Windows platformoje, bet tam galima naudoti skirtingas programavimo kalbas, kurias palaiko .Net, nes visas kodas yra kompiliuojamas į MSIL (Microsoft Intermediate Language).

RMI skirtingų platformų atžvilgiu turi pranašumą - kadangi RMI/JRMP yra paremta Java kalba, tai sistemos gali veikti bet kurioje operacinėje sistemoje, kurioje yra virtuali Javos mašina (UNIX, Windows ir t.t.). Tačiau RMI galima naudoti tik Java programavimo kalbą, ir nors tai yra apribojimas, jis turi gerąją pusę - RMI yra gana gerai optimizuotas, nes priklauso tik nuo vienos programavimo kalbos [Ste02].

RMI-IIOP protokolas suteikia sąlygas Java kalbai sąveikauti su bet kuria kita kalba (sistema) kuri veikia naudodama CORBA [Zah00]. Tai yra didžiulis pliusas RMI technologijai, daug kas jau seniai laukia panašaus sprendimo ir .Net Remoting pusėje, bet kol kas to nėra. Vienas iš privalumų naudojant RMI-IIOP yra tas, kad norint RMI programą sujungti su CORBA programa, programuotojams nereikia mokėti CORBA IDL kalbos.

3.2.5. Aktyvavimas

Java RMI objektai gali būti pažymėti kaip aktyvuojami paveldint juos iš *Activatable* klasės. Tai reiškia, kad tie objektai gali būti užregistruojami registre nesukuriant jų egzempliorių, o egzemplioriai sukuriami tik kai klientas kviečia objektą. Dar galima padaryti objektų nuorodas pastovias, kas leidžia objekto nuorodai egzistuoti, net jei pats nutolęs objektas jau nebeegzistuoja. RMI turi ir tokią sąvoką kaip aktyvacijos grupė – tai grupė sudaryta iš aktyvuojamų nutolusių objektų, kuri visa gali būti aktyvuojama kartu.
.Net Remoting objektai kaip jau minėjome anksčiau gali būti aktyvuojami serverio arba kliento, o serverio aktyvuojami objektai gali būti arba vieno egzemplioriaus arba vieno kvietimo.

3.2.6. Kiti skirtumai

Sąveikai tarp kliento ir serverio yra naudojamos skirtingos papildomos klasės. RMI atveju tai yra įgaliotiniai ir skeletai, o .Net Remoting savo ruožtu tarpines klases naudoja tik klientinėje pusė. Tačiau tarp jų yra dar vienas svarbus skirtumas, nes RMI visos papildomos klasės yra sugeneruojamos kompiliavimo metu, o Remoting atveju jos sukuriamos veikimo metu, kas gali turėti įtakos veikimo greičiui.

Dar vienas skirtumas kuris nėra tiesiogiai su sijęs su RMI ar Remoting vidiniais procesais, bet gali paveikti Remoting programos veikimo greitį yra .Net procesas vadinamas *Įpakavimu* (Boxing). Nors Java kalbos tipų sistema yra beveik objektiškai orientuota, tačiau ji vis dar naudoją kelis primityvius tipus, tokius kaip *int* ar *char*. .Net neturi primityvių tipų. Įpakavimo mechanizmas leidžia bet kokį tipą laikyti objektu, padarydamas .Net tipų sistemą visiškai objektinę, tačiau tai atsiliepia veikimo greičiui.

4. Java RMI ir .Net Remoting veikimo greičio tyrimas

Tiek RMI, tiek Remoting įneša tam tikras pridėtines vykdymo laiko sąnaudas, todėl vienas iš šio tyrimo tikslų yra išsiaiškinti būtent kokio dydžio yra tos pridėtinės sąnaudos. Kitas tikslas yra nustatyti vykdymo sulėtėjimą esant didesniems klientų užklausų srautams. Todėl buvo sukurti keli komunikacijos tarp kliento ir serverio scenarijai.

4.1. Matavimų specifikacija

Buvo iškelti tokie pagrindiniai reikalavimai veikimo greičio matavimo tyrimui:

- Rezultatai gauti iš Java RMI ir .Net Remoting turi būti palyginami.
- Turi būti simuliuojami vieno ir kelių klientų veikimas.
- Turi būti panaudoti skirtingi duomenų tipai.

- Turi būti naudojama eilinio naudotojo aplinką atitinkanti kompiuterinė įranga.

Kad būtų įmanoma įvairiapusiškai įvertinti abi technologijas, matavimams atlikti buvo pasirinktos trys strategijos:

- Nutolusių metodų iškvietimas kaip parametrą naudojant paprastus duomenų tipus.
- Nutolusių metodų iškvietimas naudojant įvairių dydžių simbolių eilutės (*String*) tipo duomenis.
- Apsikeitinėjimas objektų kolekcijomis tarp kliento ir serverio, pašalinant po vieną elementą kiekvienoje pusėje.

Pasikartojančiai kviečiant nutolusį metodą su mažu kiekiu duomenų (kaip parametru) turi paaiškėti kaip tarpinė programinė įranga (šiuo atveju galvoje turima RMI arba Remoting sluoksnį) susitvarko su grynais metodų kvietimais, nes duomenys gana maži ir smarkiai įtakoti veikimo neturėtų. Siunčiant didesnius *String* tipo duomenis turėtumėm sužinoti kaip šios technologijos susitvarko su duomenų perdavimu, o naudodami objektų sąrašus turėtumėme išbandyti sistemas ir didelio duomenų kiekio perdavimo ir šiukšlių surinkimo sąlygomis.

Vieno kliento atveju atliekami bandymai dar bus padalinti į dvi dalis. Nors realiai tokių tikrų sistemų greičiausiai nebūna (bent jau jos būtų netikslingos), bet mes atliksime bandymus kai kliento ir serverio programa veikia tame pačiame kompiuteryje ir skirtinguose.

Kelių klientų simuliavimo atveju paleisime veikti iki keturių klientų vienu metu. Iš pradžių bus tik vienas klientas, tada po vieną kliento programą iš dviejų skirtingų kompiuterių, po to viename kompiuteryje veiks dvi, o kitame tik viena klientinė programa, o galiausiai pabaigoje bus paleidžiama po 2 kliento programas kiekviename atskirame kompiuteryje.

4.2. Testavimo aplinka

Testams atlikti buvo naudojami 3 kompiuteriai: stacionarus kompiuteris, skirtas serveriui, ir du nešiojami kompiuteriai. Stacionarus kompiuteris tinklo kabeliu prijungtas prie maršrutizatoriaus, o kiti kompiuteriai buvo sujungti naudojant bevielį tinklą. Nors bevielio tinklo naudojimas gali atsiliepti vykdymo rezultatams, tačiau abiem technologijoms bus suteiktos tokios pat sąlygos Detalesnę informaciją susijusią su naudota technine ir programine įranga galite rasti 2 lentelėje.

2 lentelė. Tyrimo metu naudotų kompiuterių parametrai

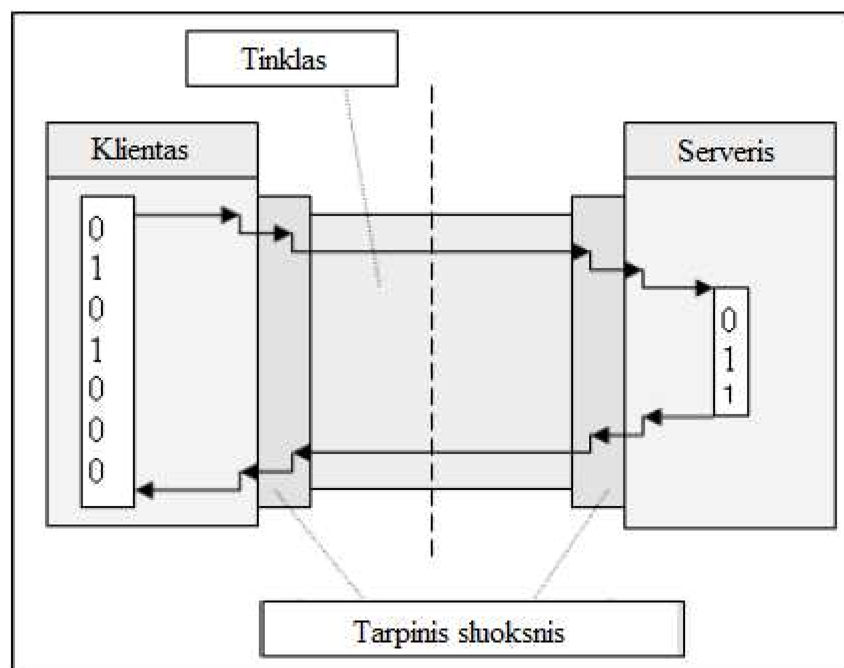
	PC1	**PC2**	**PC3**
Procesorius	Intel Pentium IV 2.4 GHz	Intel Pentium Dual-Core 1.6 GHz	Intel Pentium M 1.7 GHz
Operatyvioji	512 Mb	1.0 Gb	512 Mb

atmintis			
Bevielio tinklo sąsaja	-	802.11g	802.11g
Operacinė sistema	Windows XP	Windows Vista	Windows XP
.Net Framework versija	3.5	3.5	3.5
Java versija	1.6	1.6	1.6

Kaip matyti kompiuteriai nėra kažkokie ypatingi, todėl manau, jog ši kompiuterinė įranga tenkina mūsų pačių anksčiau apibrėžtą reikalavimą, kad testavimui turi būti naudojami kompiuteriai, atitinkantys paprasto eilinio naudotojo kompiuterį.

4.3. Matavimams naudotų programų realizacijų ypatumai

Matavimams atlikti tiek Java RMI, tiek .Net Remoting aplinkoje, iš viso buvo sukurtos keturios programos, t.y. po dvi programas, įgyvendinančias standartinę kliento-serverio sąveiką, kiekvienai iš technologijų. Nepaisant kelių C# ir Java sintaksės skirtumų, atitinkamos programos buvo beveik vienodos. 6 pav. galime matyti kaip klientas sąveikauja su serveriu, tarpinis sluoksnis paveiksle yra Java RMI arba .Net Remoting mechanizmas.



6 pav. Kliento ir serverio sąveika

Kadangi mūsų tikslas yra ištirti būtent tarpinio sluoksnio veikimo greitį ir efektyvumą, kiek įmanoma neįtraukiant viso kito programos veikimo laiko, tai laiko skaičiavimas yra paleidžiamas

prieš pat nutolusio metodo kvietimą ir sustabdomas iškart po to, kai gaunamas atsakymas iš serverio. Visi serverio objekto metodai, kurie priima vienokio ar kitokio tipo reikšmę kaip parametrą, atgal grąžina tą pačią reikšmę, išskyrus metodą, kuris priima objektų kolekciją, nes šiuo atveju grąžinama vienu elementu sumažinta kolekcija.

Visi metodų kvietimai su primityviais tipais, siekiant išvengti atsitiktinai greitų ar lėtų veikimo atvejų, buvo vykdomi po 3000 kartų, kiekvieno vykdymo laikus fiksuojant ir išsaugant masyve. Vykdymo pabaigoje būdavo suskaičiuojamas vidutinis metodo kvietimo laikas. Viskas analogiškai vyko ir atliekant bandymus su skirtingų dydžių simbolių eilučių tipo objektais, tik čia su kiekvieno ilgio objektu metodas buvo kviečiamas ne po 3000 kartų, o po 1000.

Truputį kitoks veikimo principas buvo naudojamas, atliekant bandymus su objektų kolekcijomis. Šis testas yra tarsi savotiškas PingPong žaidimas, nes klientas nusiunčia objektų kolekciją serveriui, kuris pašalina vieną objektą ir grąžina kolekciją atgal. Tada klientas pats pašalina viena objektą iš kolekcijos ir vėl siunčia tą kolekciją serverio objektui. Šis siuntinėjimasis vyksta tol, kol kolekcijoje yra nors vienas objektas. Laiko matavimas šiuo atveju irgi skiriasi. Jeigu kituose metoduose buvo skaičiuojamas vieno kreipimosi vidurkis, tai čia skaičiuojamas visas vykdymo laikas.

Tiek Java RMI, tiek .Net Remoting sluoksniai nebuvo kažkaip specialiai derinami, kad gautumėm optimaliausius įmanomus rezultatus, buvo tiesiog naudojami numatytieji gamintojų parametrai. Reikėtų tik paminėti, kad .Net Remoting atveju, programa buvo sukonfigūruota veikti TCP protokolu.

4.4. Matavimų rezultatai

Šiame skyriuje bus pateikti išsamūs atlikto tyrimo rezultatai, jų analizė, palyginimas ir interpretavimas. Nepaisant to, kad kiekvienas testas metodų kvietimus pakartodavo daug kartų, tikslumo dėlei patys testai irgi buvo vykdomi po kelis kartus, dėl to čia bus pateikti galutiniai reikšmių vidurkiai. Visi rezultatai suapvalinti iki šimtųjų dalių.

4.4.1. Vieno kliento scenarijus

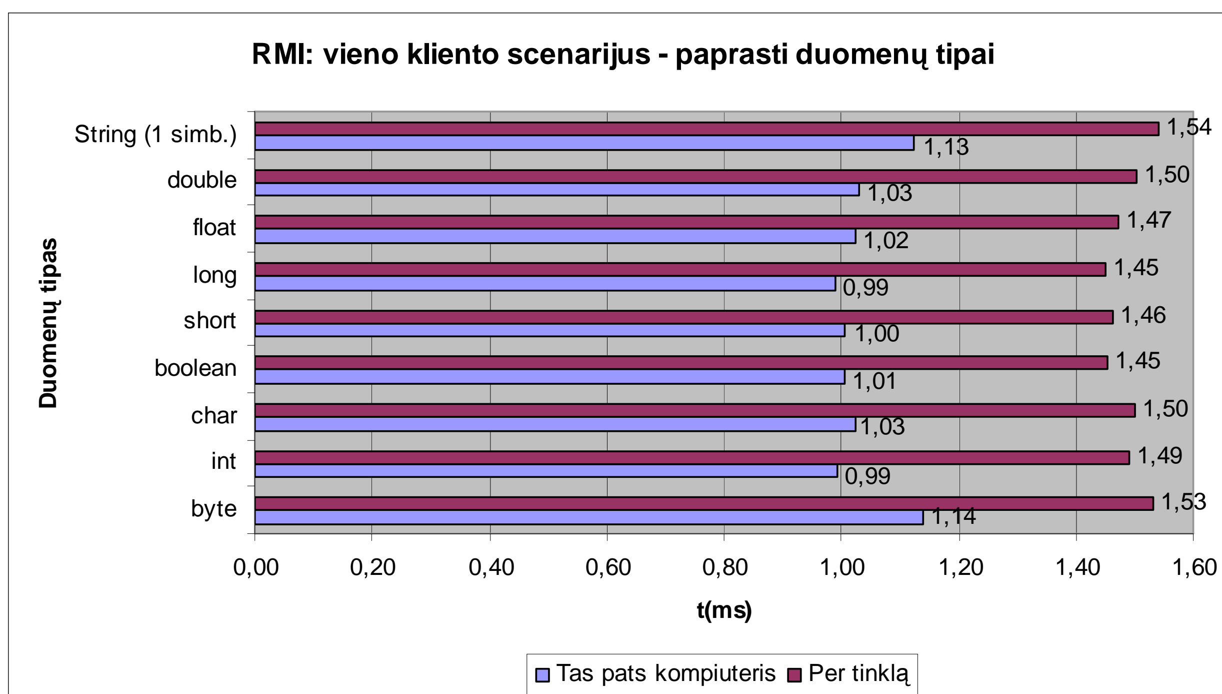
Iš pradžių testams vykdyti buvo naudojamas vienintelis klientas, kuris kvietė nutolusio serverio objekto metodus, tačiau buvo iškirti du atvejai:

- Kai klientas ir serveris veikia tame pačiame kompiuteryje.
- Kai klientas ir serveris veikia skirtinguose kompiuteriuose.

4.4.1.1. Paprasti duomenų tipai

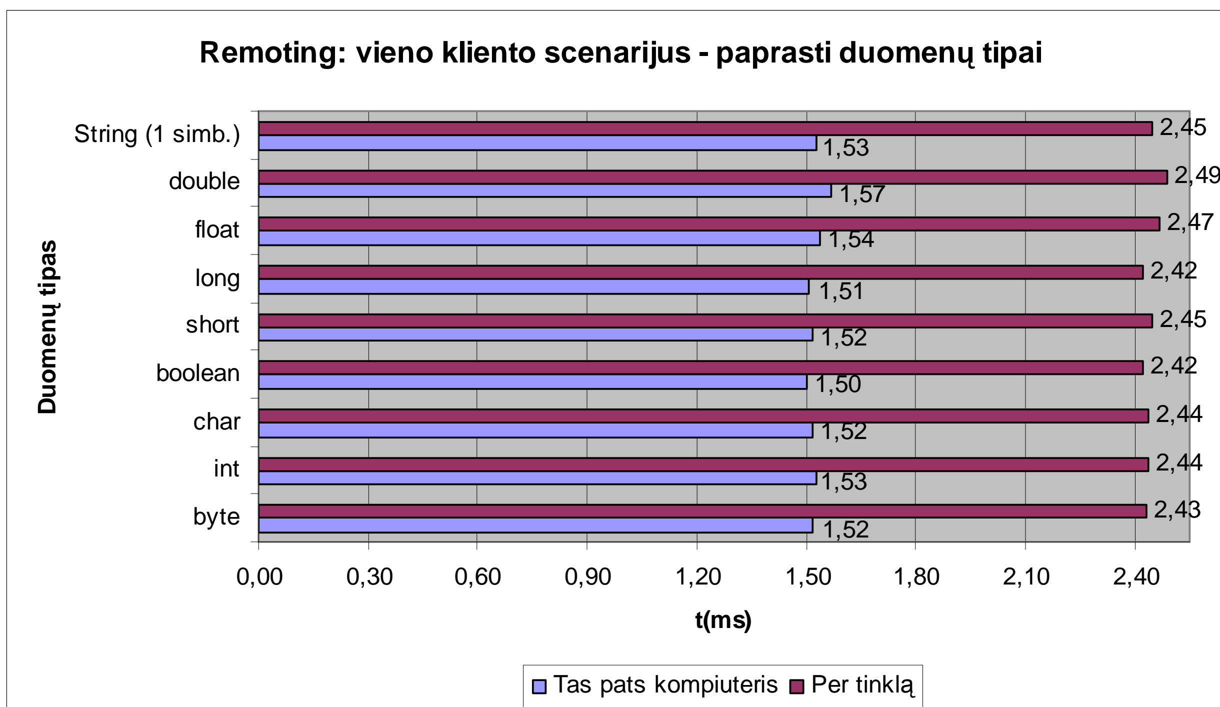
7 pav. galima matyti surinktus paprastų duomenų tipų naudojimo Java RMI aplinkoje rezultatus. Skirtumas tarp metodų kvietimo laiko, naudojant skirtingus tipus yra labai nežymus.

Pirmu atveju bendras vidurkis yra 1,03 ms, o antru atveju 1,48 ms. Taigi matome, kad veikimas per tinklą turėjo nemažą įtaką vykdymo laikui, nes jis išaugo apie 43%.



7 pav. Java RMI vieno kliento kvietimų su paprastais duomenų tipais veikimo laikas

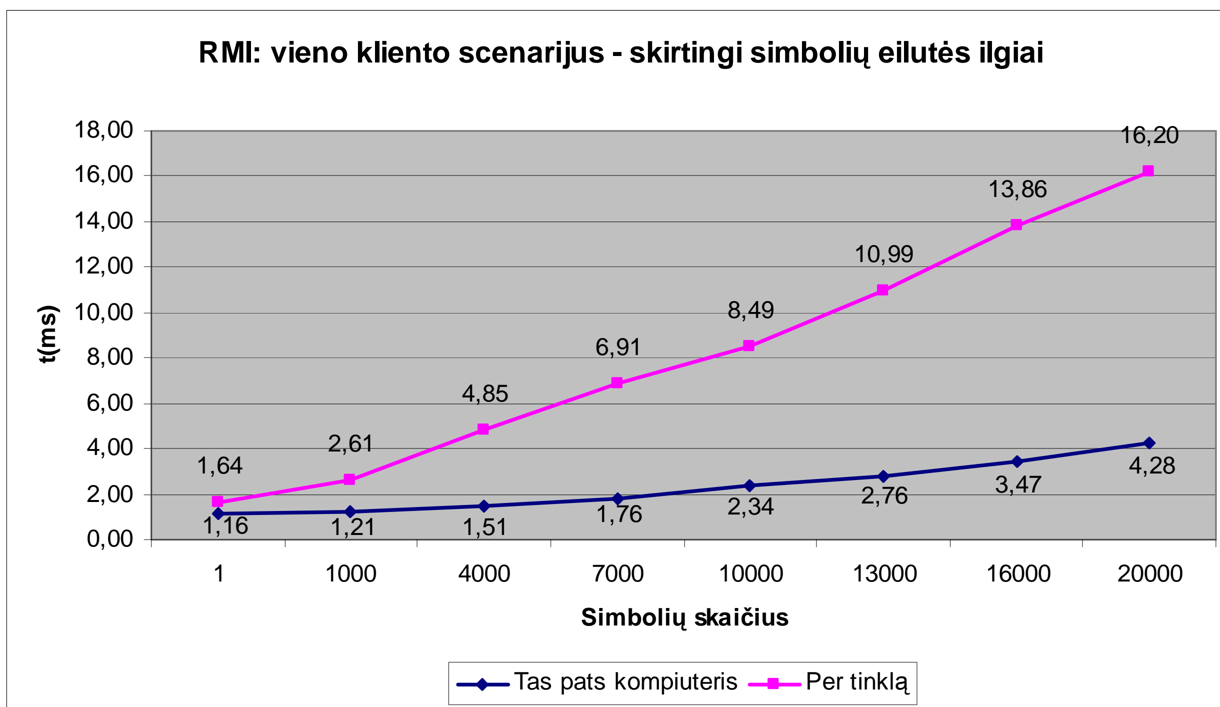
.Net Remoting atveju rezultatai panašūs tuo, kad metoduose naudojant skirtingus paprastuosius duomenų tipus vykdymo laikas skyrėsi labai nežymiai. Tačiau bendrai rezultatai labai nustebino, nes kaip matome 8 pav. net kliento ir serverio programoms veikiant tame pačiame kompiuteryje, bendras vykdymo laiko vidurkis buvo didesnis nei Java RMI veikiant per tinklą, t.y. 1,52 ms. Vykdymo laiko sulėtėjimas klientui sąveikaujant su serveriu per tinklą irgi gana didelis, nes vidutinis gautas laikas buvo 2,44 ms, o tai yra apie 60% ilgiau, nei veikiant lokaliai. Visumoje šio pirmojo testo kontekste .Net Remoting atrodo tikrai silpnai, nes bendras veikimo laikas naudojant paprastus duomenų tipus yra apie 60% didesnis nei Java RMI atveju. Kad .Net Remoting rezultatai yra blogesni, tai visiškai normalu ir greičiausiai taip yra dėl įpakavimo (boxing) proceso, tačiau buvo lauktas šiek tiek švelnesnis skirtumas, o toks gana didelis skirtumas buvo tikrai netikėtas.



8 pav. .Net Remoting vieno kliento kvietimų su paprastai duomenų tipais laikas

4.4.1.2. Skirtingi simbolių eilutės ilgiai

Antrame teste buvo tiriama siunčiamos simbolių eilutės dydžio įtaka metodo vykdymo laikui. Tyrimui pasirinktas simbolių kiekis tam tikrais intervalais kito nuo 1 simbolio iki 20000. Java RMI vykdymo rezultatai pateikiami 9 pav. Matome, kad simbolių eilutės ilgis, kaip ir buvo tikėtasi, neabejotinai turį įtakos metodų vykdymo laikui. Nors klientui ir serveriui veikiant tame pačiame kompiuteryje ta įtaka nėra labai didelė, bet Java RMI perdavimo tinklu pridėtiniai kaštai išauga gana ryškiai priklausomai nuo simbolių eilutės ilgio. Lokalaus veikimo atveju, perduodant 1000 simbolių ilgio eilutę vidutinis veikimo laikas buvo 1,21 ms, o simbolių eilutės ilgį padidinus iki 20000 simbolių veikimo laikas pailgėjo tik apie 3 ms - iki 4,28 ms. Imant tokius pat simbolių eilučių ilgius klientui ir serveriui veikiant skirtinguose kompiuteriuose, vykdymo laikas išauga nuo 2,61 ms iki 16,20 ms. Kaip matyti iš gautų skaičių ir grafiko (9 pav.), esant didesniems duomenų kiekiams, veikimas per tinklą buvo beveik keturis kartus lėtesnis.

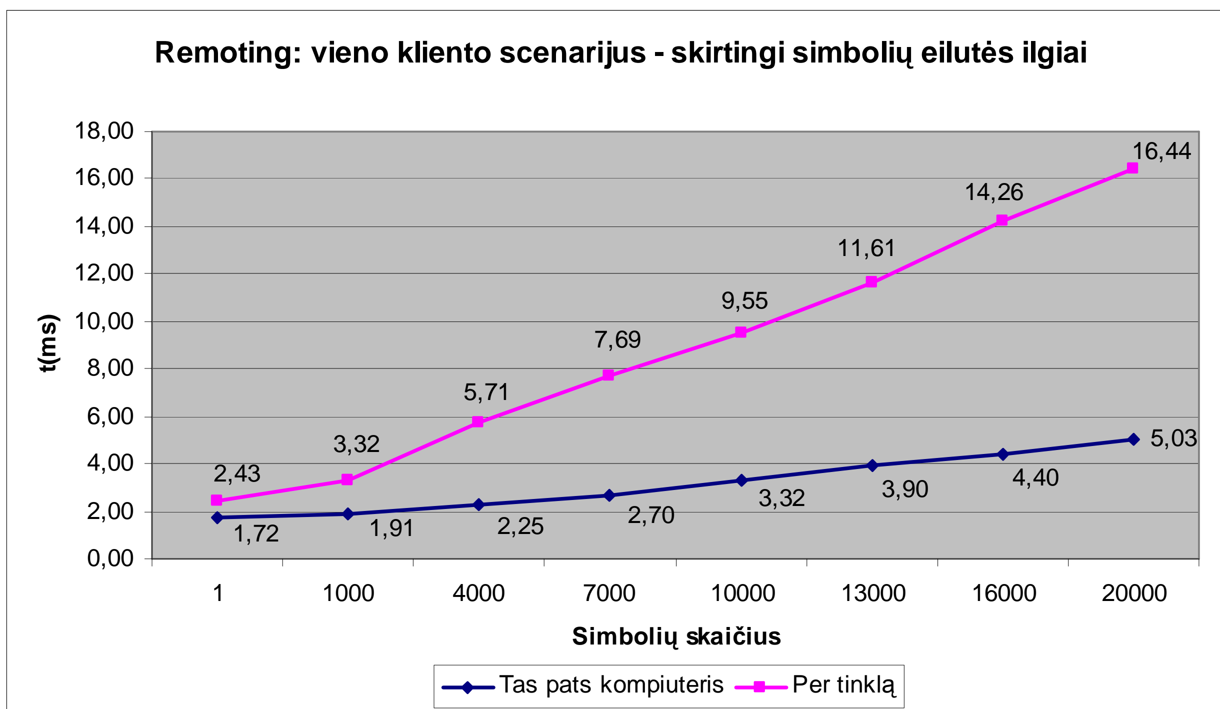


9 pav. Java RMI vieno kliento kvietimų su skirtingais simbolių eilutės dydžiais vykdymo laikas

Jei testo su paprastais duomenų tipais metu .Net Remoting rezultatai buvo prasti ir gana smarkiai atsiliko nuo Java RMI, tai šio testo parodymai kur kas optimistiškesni ir konkurencingesni. Iš 10 pav. galima matyti, kad rezultatų grafikas yra labai panašus kaip ir Java RMI atveju, tik vykdymo laikai, tiek lokalaus, tiek tinklinio veikimo atveju, vidutiniškai yra maždaug 1ms didesni, o tai reiškia, kad ir antrojo testo metu šiek tiek spartesnė visgi pasirodė Java RMI platforma. Vykdymo sulėtėjimas simbolių eilutei išaugant nuo 1000 simbolių iki 20000 yra beveik vienodas kaip ir RMI:

- veikiant tame pačiame kompiuteryje veikimo laikas išauga nuo 1,91ms iki 5,03ms (skirtumas apie 3ms)
- komunikuojant per tinklą nuo 3,32 ms iki 16,44 ms (skirtumas apie 13ms).

Kadangi, ir testo su paprastais duomenų tipais ir šio testo metu, vykdymo laikų skirtumas tarp atitinkamų Java RMI ir .Net Remoting metodų kvietimų yra labai panašus ir yra nepriklausomas nuo perduodamų duomenų kiekio, tai galima daryti išvadą, kad .Net Remoting tarpinis sluoksnis turi ilgesnę nutolusių objektų inicializavimo ir kitų paruošiamųjų darbų fazę. Lieka tikėtis, kad .Net Remoting dar parodys geresnį laiką esant didesniems duomenų srautams, kai pradinės laiko sąnaudos bus pakankamai mažos lyginant su duomenų persiuntimo laiku, nes kol kas Java RMI yra aiškus lyderis.



10 pav. .Net Remoting vieno kliento kvietimų su skirtingais simbolių eilutės dydžiais vykdymo laikas

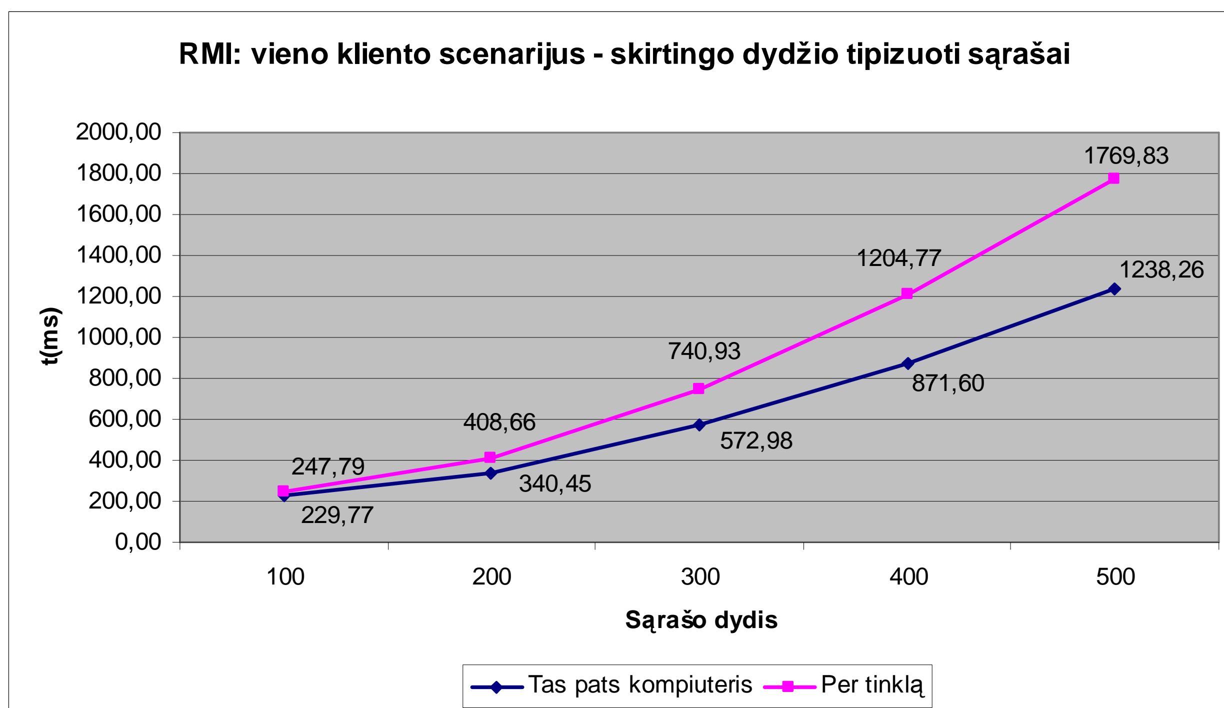
4.4.1.3. Skirtingo dydžio tipizuoti sąrašai

Trečiojo testo metu sąveikai tarp kliento ir serverio buvo naudoti tipizuoti sąrašai, kurie buvo užpildyti nesudėtingos klasės, turinčios kelias savybes ir metodus, egzemplioriais. Sąrašų dydžiai buvo pasirinkti tokie: 100, 200, 300, 400 ir 500 elementų. Taigi kiekvieno atvejo metu įvykdavo būtent tiek kvietimų tarp kliento ir serverio, kokio ilgio būdavo pats sąrašas, nes siuntinėjimasis pirmyn-atgal vykdavo tol, kol sąraše nelikdavo elementų.

Kaip galima matyti 11 pav., šio tyrimo Java RMI aplinkoje grafikas nėra tiesinis, nes vykdant testus, esant skirtingiems sąrašo ilgiams, kinta ne vien perduodamų duomenų kiekis, bet ir metodų kvietimų skaičius. Veikiant tame pačiame kompiuteryje vykdymo laikas, pradiniam sąrašo ilgiui esant 100 elementų, buvo 229,77 ms, o toliau elementų kiekį didinant po 100, veikimo laikas atitinkamai vis pailgėja maždaug: 110, 230, 300 ir 370 milisekundžių. Taigi matome, kad sąrašo ilgį didinant vienodu kiekiu elementų, veikimo laikas kinta ne tolygiai – kuo pradinis sąrašo ilgis tampa didesnis, tuo veikimo laiko pasikeitimas didesnis.

Veikimo per tinklą rezultatų grafikas yra gana panašus kaip ir veikiant lokaliai, tik greičiau kyla aukštyn, nes visos reikšmės yra atitinkamai didesnės dėl duomenų perdavimo tinklu. Vykdymo laikas sąrašui turint 100 elementų buvo 247,79 ms, o tai yra tik 18 ms daugiau nei veikiant lokaliai, tačiau sąrašą vis didinant po 100 elementų, veikimo sulėtėjimas yra didesnis t.y. maždaug: 160, 330, 460 ir 560 milisekundžių. Lyginant su atitinkamomis lokalaus veikimo sulėtėjimo reikšmėmis, veikimo per tinklą sulėtėjimas yra apie 1,5 karto didesnis (pvz. sąrašo

elementų skaičių padidinus nuo 400 iki 500, lokalaus veikimo atveju vykdymas sulėtėja 370ms, o per tinklą 560). Kadangi veikiant per tinklą kiekviename žingsnyje sulėtėjimas yra apie 1,5 didesnis, tai sąrašui turint 500 elementų bendras vykdymo laikas buvo 1769,83 ms ir tai yra jau maždaug 530 ms daugiau nei veikiant lokaliai.

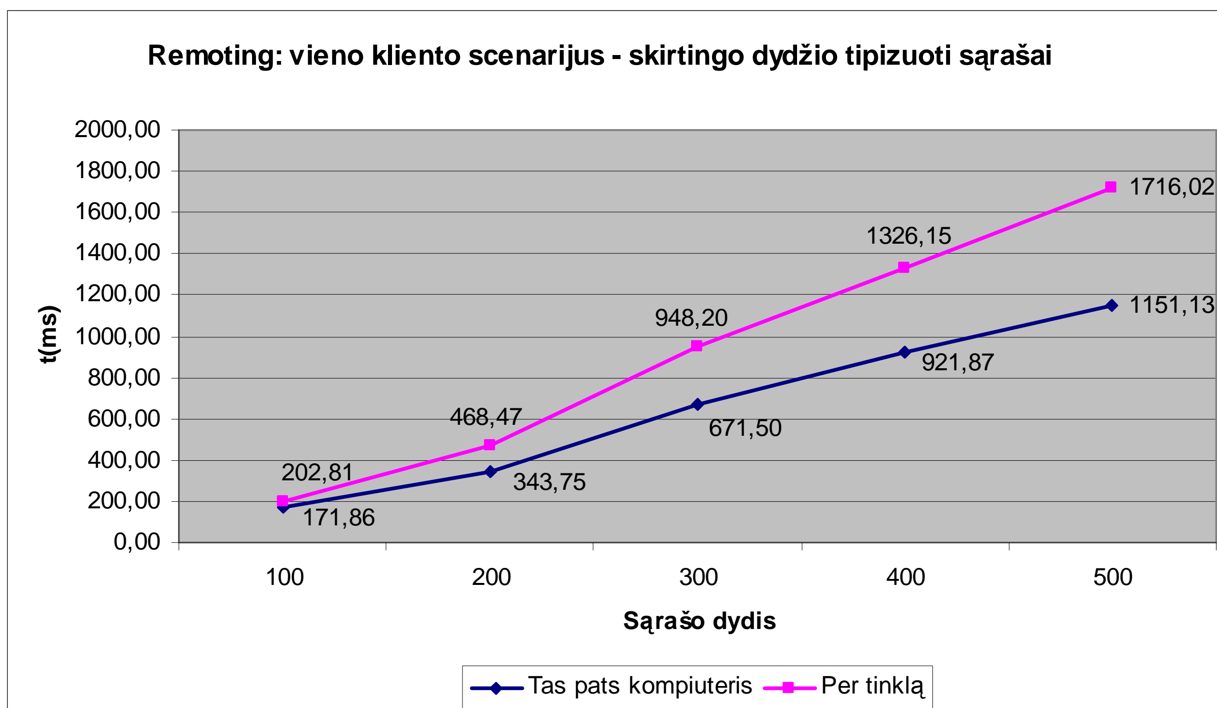


11 pav. Java RMI vieno kliento kvietimų su skirtingais tipizuoto sąrašo dydžiais vykdymo laikas

Pažvelgę į nutolusių metodų kvietimų, naudojant tipizuotus sąrašus .Net Remoting aplinkoje, rezultatus 12 pav., matome, kad pradiniam sąrašui turint 100 elementų, vykdymo laikas, tiek veikiant lokaliai, tiek per tinklą, yra mažesnis nei Java RMI (atitinkamai 171,86 ms ir 202,81 ms), o tai pagaliau įneša šiokio tokio konkurencingumo į šį tyrimą. Žodžiai „šiokio tokio" praeitame sakinyje panaudoti neatsitiktinai, nes pradinį sąrašą padidinus iki 200 elementų, .Net Remoting vėl atsilieka, nes vykdymo laikas lokaliu atveju išauga dvigubai – iki 343,75 ms, o veikiant per tinklą dar daugiau t.y. apie 2,3 karto – iki 468,47 ms. Tačiau tai dar ne viskas, nes kai sąrašo dydis yra 500 elementų, Java RMI rezultatai vėl yra prastesni. Šiuo atveju .Net Remoting veikiant viename kompiuteryje užtruko 1716,02 ms – beveik 54ms greičiau, o veikimo per tinklą laikas buvo 1151,13ms ir tai yra beveik 90ms greičiau.

Nors šio tyrimo metu .Net Remoting prie tam tikrų sąrašo ilgių ir parodė geresnius rezultatus, tačiau tas greitesnis veikimas nėra pastovus. Jei Java RMI atveju keičiant sąrašo apimtį pastoviu dydžiu t.y. 100 elementų vykdymo sulėtėjimas visada tik didėdavo, tai .Net Remoting atveju jis yra tarsi banguojantis. Klientui ir serveriui veikiant tame pačiame kompiuteryje gaunamos tokios apytikslės sulėtėjimo reikšmės: elementų kiekį padidinus nuo 100 iki 200 - 170, nuo 200 iki 300 – 330, nuo 300 iki 400 – 250, nuo 400 iki 500 – 230 ms. Klientui

ir serveriui veikiant per tinklą atitinkamai maždaug: 265, 480, 380 ir 390 ms. Vėlgi, nepaisant šiokių tokių svyravimų, veikimo lokaliai ir per tinklą sulėtėjimo skirtumą išreiškę kartais, kaip ir Java RMI atveju, gauname apie 1,5 karto.



12 pav. .Net Remoting vieno kliento kvietimų su skirtingais tipizuoto sąrašo dydžiais vykdymo laikas

Kadangi šio testo lyderis nėra aiškus, nors statistiškai, truputį didesnę laiko dalį greičiau veikė Java RMI, vėliau buvo nuspręsta atlikti kelis papildomus nedidelius tyrimus tam kad išsiaiškinti kaip veikia .Net Remoting platesniame reikšmių diapazone (žiūrėti skyrių 4.5).

4.4.2. Kelių klientų scenarijus

Nors iš jau gautų ir matytų rezultatų galima įvardinti greitesnę technologiją, tačiau tai buvo tik vieno kliento scenarijaus rezultatai, o išskirstytos sistemos dažnai būna sukurtos pagal tokią struktūrą, kad su serveriu vienu metu bendrauja kelios, keliolika ar net keli tūkstančiai klientų. Gali būti, kad technologija veikianti greitai su vienu klientu gali pradėti veikti žymiai lėčiau klientų skaičiui padidėjus. Šiame skyriuje bus pateikti tyrimo rezultatai, kurio metu buvo atliekami tokie patys testai kaip ir vieno kliento atveju, tik į serverį vienu metu kreipdavosi iki keturių klientų (veikiančių per tinklą).

Kadangi į toliau pateiktus grafikus skaitinės reikšmės netelpa, o pačios diagramos yra gana informatyvios ir suprantamos, tai visos susiję reikšmių lentelės pateiktos prieduose.

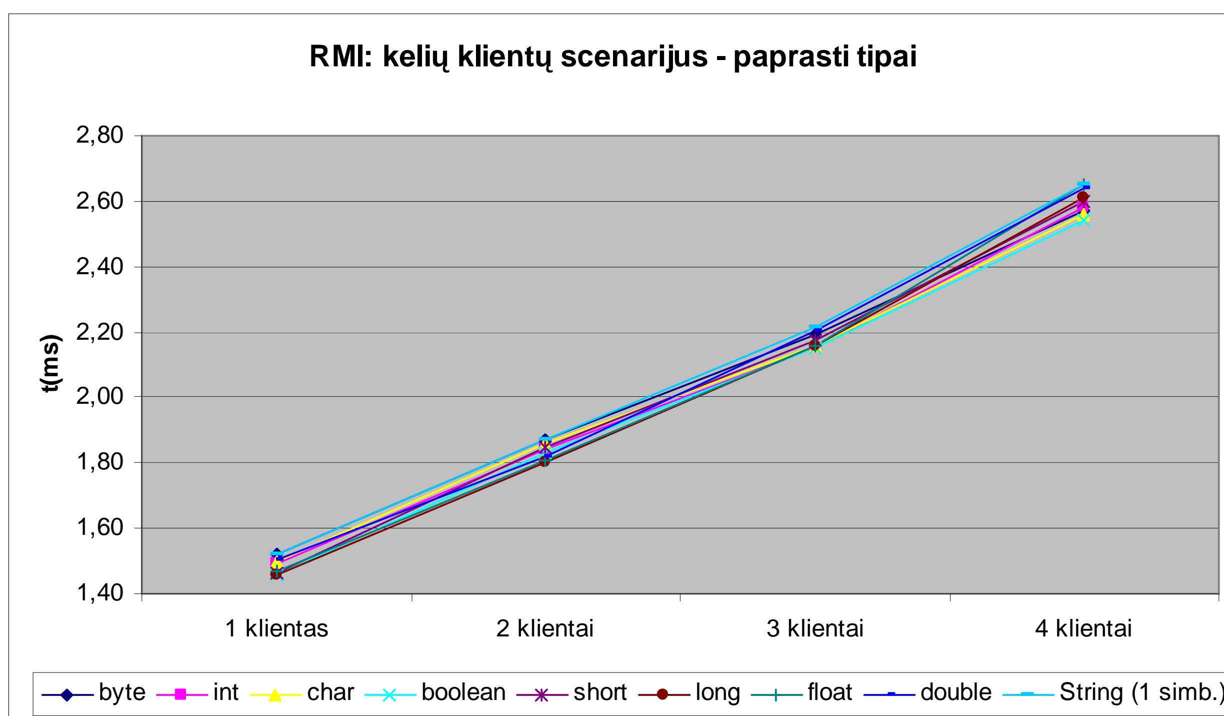
4.4.2.1. Paprasti duomenų tipai

Kadangi kaip jau nustatyta anksčiau, laiko skirtumas tarp metodų kvietimų su paprastaisiais duomenų tipais yra mažas, tai šio skyrelio abiejuose grafikuose, tiesės, atitinkančios skirtingus duomenų tipus, yra labai arti viena kitos ir tarsi sudaro vieną storą liniją, todėl visus šiuos tipus aptarsime bendrai - kaip vieną visumą.

Java RMI šiame kontekste parodė tikrai neblogus rezultatus, nes, kaip galima matyti 13 pav., klientų skaičiui išaugus nuo 1 iki 4 t.y. padidėjus 4 kartus, vykdymo laikas teišauga apytiksliai 1,7 karto, nuo ~1,48 ms iki ~2,6 ms. Iš turimo grafiko, truputį suapvalinus kai kuriuos duomenis, galima išvesti tokią formulę:

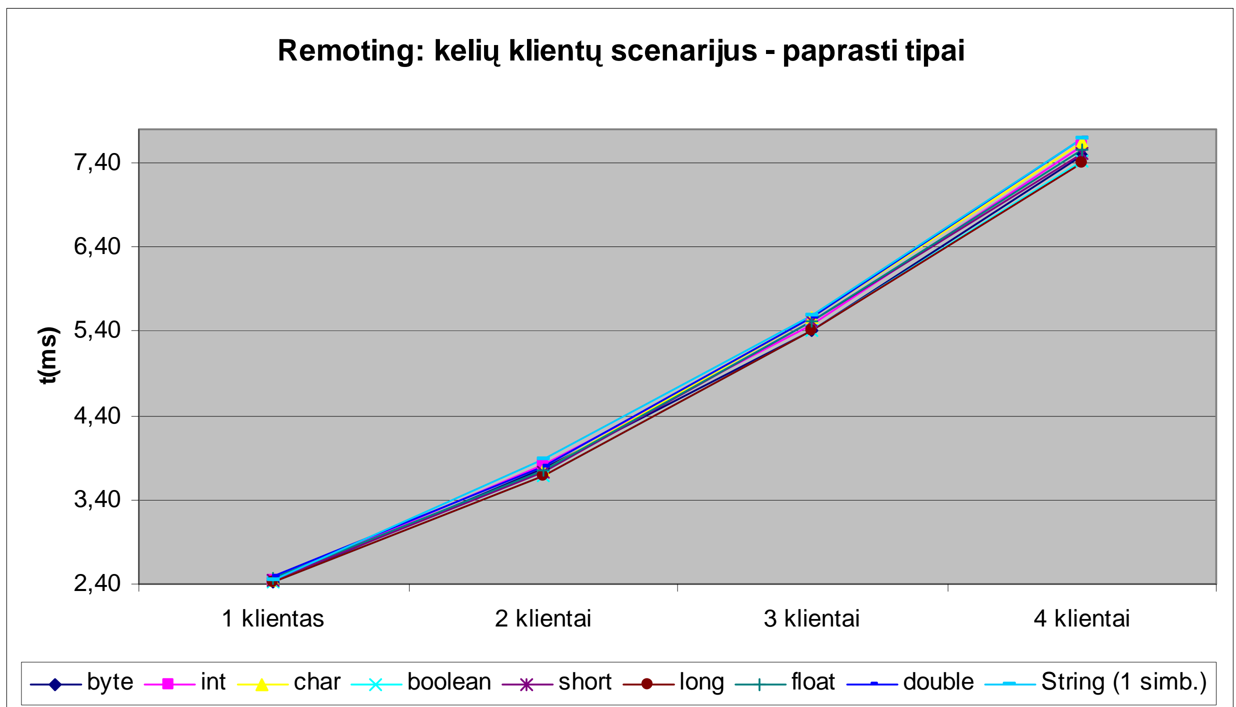
$$t(ms) = 0,4k + 1$$

kur t yra vykdymo laikas milisekundėmis, o k – klientų skaičius. Taigi matome kad kiekvienas papildomas klientas veikimo laiką pailgina 0,4 ms.



13 pav. Java RMI kelių klientų kvietimų su paprastais duomenų tipais vykdymo laikas

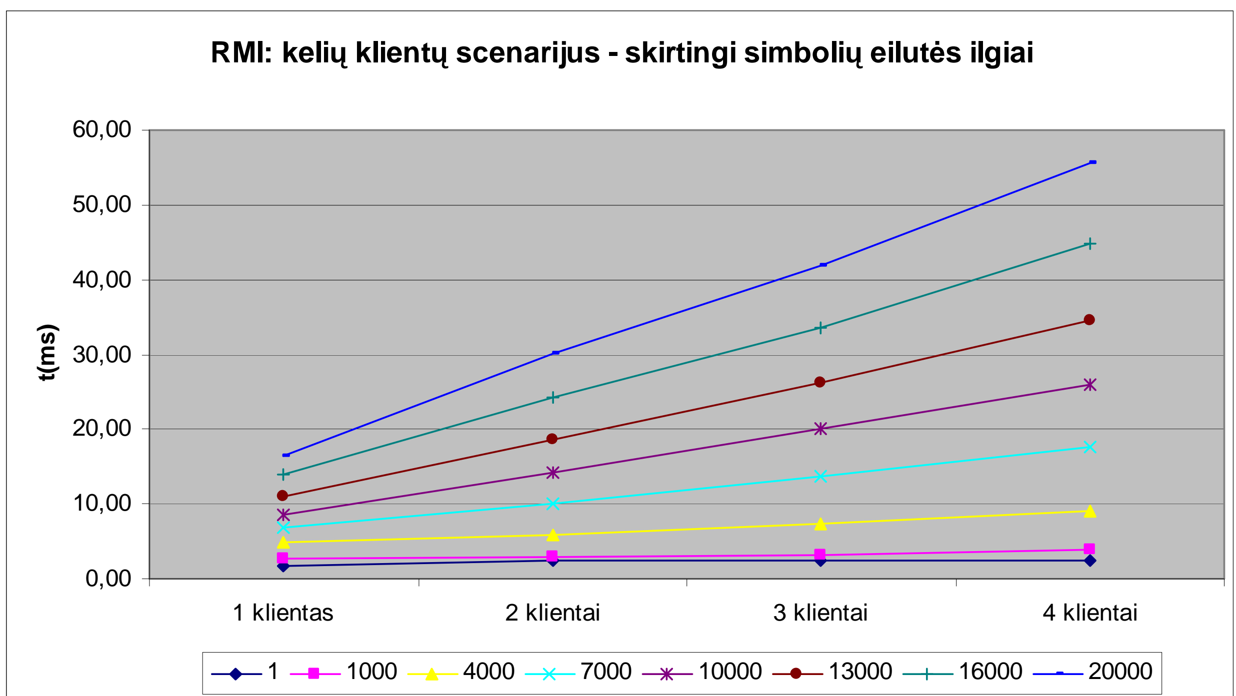
Jeigu vieno kliento atveju su paprastais duomenų tipais .Net Remoting parodė gan prastus rezultatus, tai kelių klientų veikimas tuos rezultatus dar labiau pablogino. Nors 14 pav. esantis grafikas vizualiai yra labai panašus į prieš tai buvusį, nereikia apsigauti, nes šio grafiko reikšmių sritis yra žymiai platesnė. Java RMI grafiko reikšmių srities plotis buvo tik 1.4 ms (nuo 1.4 ms iki 2.8 ms), o šiuo atveju jis yra net 5 ms. Kadangi pradinis laikas esant vienam klientui yra tik 0,2 ms mažesnis nei Java RMI veikimo su keturiais klientais rezultatas, o vykdymo laiko skirtumas tarp 1 ir 4 klientų yra 5 ms (maždaug 3 kartai) , tai .Net Remoting čia yra totalus autsaideris.



14 pav. .Net Remoting kelių klientų kvietimų su paprastais duomenų tipais vykdymo laikas

4.4.2.2. Skirtingi simbolių eilutės ilgiai

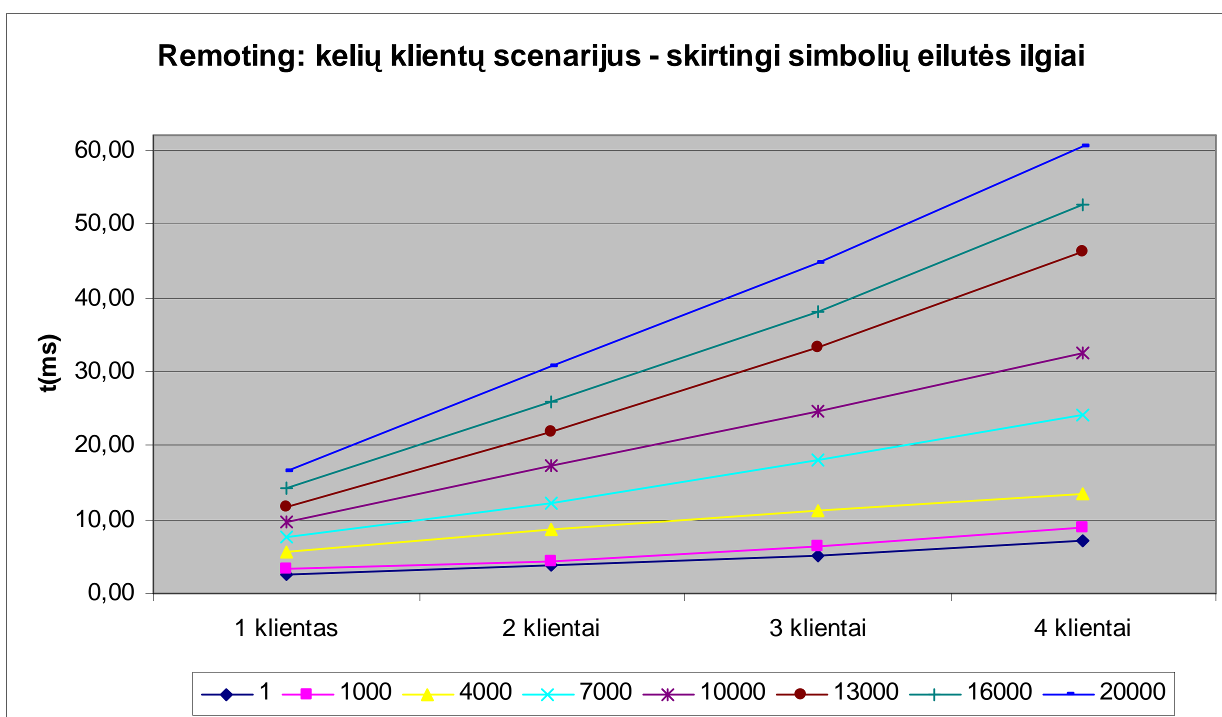
Kaip matome 15 ir 16 pav. pats klientų skaičiaus padidėjimas labai didelės įtakos vykdymo laikui neturi, nes esant mažiems duomenų kiekiams jis kinta nežymiai, be to, tie kelių klientų kvietimai gali ir visai prasilenkti. Tačiau kuo didesnis perduodamų duomenų kiekis, tuo didesnę įtaką turi vienu metu veikiančių klientų skaičius.



15 pav. Java RMI kelių klientų kvietimų su skirtingais simbolių eilučių ilgiais vykdymo laikas

Jei Java RMI atveju (žiūrėti 15 pav.), klientų skaičius padidėja nuo 1 iki 4, tai simbolių eilutei turint 1000 elementų veikimo laikas pailgėja tik 1,32 ms, 10000 – 17,5 ms, o eilutės ilgiui siekiant 20000 vykdymo pailgėjimas jau beveik 40 ms.

Ne kokius rezultatus visuose testuose parodžiusiai .Net Remoting technologijai, čia buvo bene paskutinė proga parodyti, kad ji sugeba nors kažką daryti greičiau nei Java RMI, tačiau pažiūrėjus į 16 pav. atsiskleidžia kitokia niūri tiesa. Nors grafikas iš pirmo žvilgsnio vėl labai panašus į Java RMI grafiką, ir esant 1 klientui vizualiai įžiūrėti skirtumus tarp grafikų sunku, tačiau pradedant dviem klientais, plika akimi galima matyti, kad vykdymo laikai tikrai yra didesni. .Net Remoting atveju klientų skaičiui padidėjus nuo 1 iki 4, o simbolių eilutei esant 1000 simbolių, vykdymo laikas pailgėja 7,5 ms, 10000 – 22,9 ms, o ties 20000 – 44 ms. Taigi įvertinus ir tai, kad pradinis laikas esant vienam klientui .Net Remoting irgi didesnis nei Java RMI, tai galutinis rezultatas ir vėl ne koks.



16 pav. .Net Remoting kelių klientų kvietimų su skirtingais simbolių eilučių ilgiais vykdymo laikas

Kadangi tipizuotų sąrašų atvejis, veikiant keliems klientams, yra beveik identiškas ką tik aprašytiems pavyzdžiams su simbolių eilutėmis (skiriasi tik reikšmių sritis ir proporcijos), o įdomesnis variantas, kai esant vienam klientui .Net Remoting su 100 ir 500 elementų sąrašais buvo greitesnis už Java RMI irgi jau išnagrinėtas prie vieno kliento scenarijaus, tai čia smarkiai nebeišsiplėsime. Reikia tik paminėti, kad tiek dviejų, tiek trijų, tiek keturių klientų atveju .Net Remoting visais atvejais atsiliko nuo Java RMI, o veikimo greičio sumažėjimas irgi buvo didesnis.

4.5. Papildomi tyrimai

Kadangi vieno kliento atveju, kviečiant metodus su skirtingo ilgio simbolių eilutėmis, .Net Remoting atsilikimas nuo Java RMI visą laiką buvo daugmaž vienodas, o simbolių eilutei esant 20000 ilgio, tas atsilikimas sumažėjo (žiūrėti 9 pav., 10 pav.). Kyla natūralus klausimas – o kas dedasi už tos ribos? Gal .Net Remoting vykdymo laikas dar sumažėja ir net aplenkia Java RMI? Panašūs klausimai kyla ir žiūrint metodų kvietimų, naudojant tipizuotus sąrašus, veikimo laiko grafikus (žiūrėti 11 pav., 12 pav.). Kas būtent vyksta už tos 500 elementų ribos? Ar toliau jau .Net Remoting greitesnis už Java RMI?

Tam, kad gautumėm atsakymus į šiuos klausimus, buvo nuspręsta atlikti, pora papildomų tyrimų, praplečiant tiek simbolių eilutės dydį, tiek sąrašo elementų skaičių. Simbolių eilutės ilgiai panašiais intervalais buvo pratęsti iki 35000 simbolių, o sąrašo dydžiai kas šimtą pratęsti iki 1500 elementų.

Atlikus testus ir palyginus rezultatus, tiek naudojant su simbolių eilutes, tiek tipizuotus sąrašus išaiškėjo įdomus reiškinys. Tekste jau buvo minėta, kad .Net Remoting rezultatai yra tarsi truputį banguojantys, būtent tą ir patvirtino šis papildomas tyrimas. Buvo nustatyta, kad prie tam tikrų sąlygų t.y. esant tam tikram simbolių eilutės ar sąrašo ilgiui, Remoting programos vykdymo laikas padidėja staigiau nei įprastai, o po to seka mažesni vykdymo laiko padidėjimai. Taigi kartais rezultatams taip subangavus .Net Remoting priartėja arčiau prie Java RMI, ar net sugeba parodyti geresnį laiką, tačiau tai įvykdavo palyginti retai, o įžvelgti kažkokio logiško tų svyravimų dėsningumo irgi nepavyko (aišku kažkiek svyruoja ir Java RMI rezultatai, bet .Net Remoting atveju tie svyravimai būna žymiai ryškesni).

Rezultatai ir išvados

Pirmoje darbo dalyje išsiaiškinta kas tai yra išskirstytos sistemos, kokie jų privalumai, išnagrinėti jų veikimo principai, kas leidžia žymiai geriau suprasti, kam išskirstytos sistemos naudojamos ir kaip jos veikia. Išanalizuota technologijų raida ir įvairovė padeda susidaryti geresnį vaizdą nuo ko viskas prasidėjo ir iki ko išsivystė, kuo vienos technologijos skiriasi nuo kitų, kodėl vienos populiaresnės, o kitos ne ir t.t.

Toliau darbe atlikta Java RMI ir .Net Remoting t.y. dviejų šiuo metu populiarių ir tarpusavyje konkuruojančių platformų, skirtų išskirstytų sistemų kūrimui, analizė. Ištirtos jų architektūros ir veikimo principai, pateikti paprastų sistemų pavyzdžiai ir joms paleisti reikalingi žingsniai, leidžia geriau suprasti, kaip abi šios technologijos veikia. Nors tiek Java RMI, tiek .Net Remoting pagrindinis veikimo principas yra gana panašus, kaip paaiškėjo atlikus detalų palyginimą, jos turi daugiau skirtumų negu panašumų. Įvertinus visą analizės medžiagą ir atsižvelgus į tai, kad .Net Remoting pagalbines klases kompiliuoja veikimo metu, palaiko daug programavimo kalbų, neturi paprastų tipų ir juos pateikia per įpakavimo procesą, buvo galima nuspėti, kad .Net Remoting veikimo greitis turėtų būti mažesnis.

Norint įsitikinti, kuri iš nagrinėtų technologijų yra tikrai greitesnė, buvo atliktas jų veikimo greičio tyrimas, naudojant savo sukurtus testus. Testai buvo pasirinkti tokie, kad galėtumėme visapusiškai išbandyti abi technologijas, nutolusiems objektams perduodant įvairius duomenų tipus, tiek vieno, tiek kelių lygiagrečiai veikiančių klientų atžvilgiu. Visi testai buvo vykdomi testavimo aplinkoje sudarytoje iš trijų kompiuterių, taip paleidžiant iki keturių klientinių programų lygiagrečiai. Kad būtų išvengta momentinių veikimo laiko pagreitėjimų ar sulėtėjimų, testai buvo vykdomi daug kartų, skaičiuojant bendrą vidurkį. Dauguma gautų matavimų rezultatų pateikti diagramų pavidalu, kurias išanalizavus paaiškėjo, kad beveik visų sukurtų testų metu, geresnį vykdymo laiką parodė Java RMI. Naudojant simbolių eilutes ir tipizuotus objektų sąrašus rezultatai skyrėsi palyginus gana nesmarkiai, tačiau dėl jau minėto fakto, kad .Net paprastus duomenų tipus pateikia per įpakavimo procesą, Remoting naudojant juos demonstravo labai prastus rezultatus. Taip pat paaiškėjo, kad Java RMI geriau susitvarko ir su krūviu, nes vienu metu veikiant kelioms klientinėms programoms, Java RMI veikimo sulėtėjimas buvo mažesnis. .Net Remoting vieno testo metu buvo sukėlęs tam tikrų abejonių, nes buvo užfiksuotas veikimo pagreitėjimas, kuris neleido daryti galutinės išvados, neatlikus papildomų tyrimų su didesniais duomenų kiekiais. Papildomas tyrimas ir analizė parodė, kad .Net Remoting veikimo greitis, su tam tikrais duomenų kiekiais, svyruoja, tačiau pagreitėjimų būna tiesiog per mažai, kad bendras rezultatas galėtų aplenkti Java RMI.

Įvertinus visus rezultatus vienareikšmiškai atsakyti į klausimą, kuri iš pasirinktų išskirstytų sistemų technologijų yra geresnė – neįmanoma, nes daug kas priklauso nuo konkrečios aplinkos ir situacijos. Vertinant vien tik veikimo greitį, geresnė yra Java RMI, tačiau vertinant pastangas ir žingsnius reikalingus nesudėtingai sistemai sukurti aiškiau ir patogiau yra naudoti .Net Remoting. Jei sistema kuriama Unix šeimos platformoms, tai reikėtų pasirinkti Java RMI, bet jei tarkim yra poreikis naudoti skirtingas programavimo kalbas atskiroms sistemos dalims, tam tikslui tinka .Net Remoting. Įvairių aplinkybių ir sąlygų gali būti labai daug, todėl galutinis vertinimas ir technologijos pasirinkimas priklausytų būtent nuo jų.

Šaltinių sąrašas

[Bar02] T.Barnaby. Distributed .NET Programming in C#, APress, 2002.

[Bir95] K.P.Birman. Building Secure and Reliable Network Applications, Cornell University, 1995
[žiūrėta 2008-03-14]. Prieiga per internetą:
<http://citeseer.ist.psu.edu/cache/papers/cs/16139/http:zSzzSzwww.cs.cornell.edu zSzkenzSzbook.pdf/birman96building.pdf>

[Brio98] J.P.Briot. Concurrency and distribution in object-oriented programming. ACM Computing Surveys, psl. 291-329, 1998 rugsėjis.

[GJT94] G.Coulouris, J.Dollimore, T.Kindberg. Distributed Systems -- Concepts and Design, second edition, Addison-Wesley, 1994.

[Gro01] W.Grosso. Java RMI, O‘Reilly & Associates, 2001.

[JZ05] Weijia Jia, Wanlei Zhou. Distributed Systems – from concepts to implementations, Springer Science, 2005.

[Mac03] M.MacDonald. Distributed Applications: Integrating XML Web Services and .NET Remoting, Microsoft Press, 2003.

[MNW03] S.McLean, J.Naftel, K.Williams. Microsoft .NET Remoting, Microsoft Press, 2003.

[PRP06] A.Puder, K.Romer, F.Pilhofer. Distributed Systems Architecture: A Middleware Approach, Elsevier Inc., 2006

[PSZ93] P.Pili, R.Scateni, G.Zanetti. A Distributed-Integrated Medical Imaging System, Scientific visualization group, 1993

[RBS05] E.Roman, G.Brose, R.P.Sriganesh. Mastering Enterprise JavaBeans Third Edition, Wiley Publishing Inc., 2005

[Ste02] M.Steen, A.Tanenbaum. Distributed Systems Princeiples and Paradigms, Prentice-Hall Inc., 2002

[STK02] J.Snell, D.Tidwell, P.Kulchenko. Programming Web Services with SOAP, O‘Reilly & Associates, 2002

[Tha99] T.L.Thai. Learning DCOM, O‘Reilly & Associates, 1999

[Zah00] R.Zahavi. Enterprise Application Integration with CORBA Component and Web-Based Solutions, John Wiley & Sons Inc., 2000

Priedai

1 Priedas. Java RMI kelių klientų duomenys (paprasti duomenų tipai)

	1 klientas	2 klientai	3 klientai	4 klientai
byte	1,53	1,87	2,19	2,59
int	1,49	1,84	2,15	2,58
char	1,50	1,86	2,16	2,56
boolean	1,45	1,83	2,15	2,54
short	1,46	1,85	2,175	2,6
long	1,45	1,8	2,155	2,61
float	1,47	1,81	2,16	2,65
double	1,50	1,82	2,205	2,69
String (1 simb.)	1,54	1,87	2,235	2,7

2 Priedas. .Net Remoting kelių klientų duomenys (paprasti duomenų tipai)

	1 klientas	2 klientai	3 klientai	4 klientai
byte	2,43	3,78	5,4	7,5
int	2,44	3,82	5,47	7,6
char	2,44	3,75	5,554	7,64
boolean	2,42	3,69	5,42	7,43
short	2,45	3,73	5,525	7,52
long	2,42	3,68	5,4	7,41
float	2,47	3,76	5,51	7,55
double	2,49	3,79	5,57	7,68
String (1 simb.)	2,45	3,89	5,58	7,7

3 Priedas. Java RMI kelių klientų duomenys (skirtingi simbolių eilutės dydžiai)

	1 klientas	2 klientai	3 klientai	4 klientai
1	1,64	2,44	2,35	2,57
1000	2,61	2,82	3,175	3,93
4000	4,85	5,77	7,295	9,02
7000	6,91	10,04	13,74	17,64
10000	8,49	14,21	20,03	25,98
13000	10,99	18,55	26,14	34,53
16000	13,96	24,15	33,435	44,92
20000	16,30	30,16	41,78	55,6

4 Priedas. .Net Remoting kelių klientų duomenys (skirtingi simbolių eilutės dydžiai)

	1 klientas	2 klientai	3 klientai	4 klientai
1	2,43	3,79	5,15	7,06
1000	3,32	4,39	6,4	9,01
4000	5,71	8,55	11,285	13,52
7000	7,69	12,3	17,93	24,16
10000	9,55	17,34	24,595	32,45
13000	11,61	21,92	33,195	46,27
16000	14,26	25,88	38,205	52,53
20000	16,44	30,87	44,66	60,45