Porazdeljevanje bremena v CORBA okolju

Jaka Močnik

Diplomska naloga
iz
RAČUNALNIŠTVA - SMER PROGRAMSKA OPREMA

predložena
Fakulteti za računalništvo in informatiko
Univerze v Ljubljani
kot delna izpolnitev pogoja za pridobitev naslova
univerzitetni diplomirani inženir računalništva

April 2001

Mentor:

prof. dr. Borut Robič

Besedilo je oblikovano s sistemom za stavljenje LaTeX.

Barbari. Ker jê...

PORAZDELJEVANJE BREMENA V CORBA OKOLJU

Jaka Močnik

Mentor: prof. dr. Borut ROBIČ

Povzetek

Tipične objektno usmerjene aplikacije so sistemi, sestavljeni iz velikega števila med seboj komunicirajočih objektov, ki izvajajo zaporedje različnih nalog. Učinkovitost takšnih aplikacij lahko izboljšamo s preprosto paralelizacijo izvajanja neodvisnih nalog. V pričujočem delu bomo predstavili objektni model CORBA, ki skrije izvedbene podrobnosti in lokacijo objektov pred programerjem, ter opisali metode za porazdeljevanje bremena. V okviru CORBE bomo razvili sistem za porazdeljevanje objektov po vozliščih računalniškega omrežja, ki lahko uporablja različne porazdeljevalne postopke in knjižnico, ki olajša uporabo le-tega v jeziku C. Sistem bomo preizkusili na problemu iskanja mrežne topologije $N \times M$ vozlišč stopnje D z najmanjšo vsoto maksimalne in povprečne razdalje med vsemi pari vozlišč ter preučili rezultate.

LOAD DISTRIBUTION IN A CORBA ENVIRONMENT

Jaka Močnik

Supervisor: prof. dr. Borut ROBIČ

Abstract

Typical object oriented applications are systems consisting of a large number of objects, communicating among themselves and performing distinct tasks. The efficiency of such an application can be improved by performing independent tasks in parallel. In this work the CORBA object model that obscures implementation details and location of objects from the programmer is presented and an introduction to load distribution algorithms is given. A system for object distribution based on CORBA and a C language library for its use are developed. The library is then tested on the problem of finding the interconnection topology in an $N \times M$ mesh of degree D that yields minimal sum of maximum and average distance among all pairs of nodes. Finally, results of the tests are analysed.

Kazalo

Slike

1 Povzetek

Objektno usmerjeno programiranje omogoča učinkovitejši razvoj obsežnih aplikacij: približa načrtovanje aplikacije človeškemu načinu razmišljanja ter olajša ponovno uporabo kode, saj so obstoječe aplikacije sestavljene iz množice objektov, ki so samostojni in opravljajo dobro določene naloge. Oboje omogoča hitrejši razvoj.

Določene dele objektno usmerjene aplikacije lahko paraleliziramo na naraven način: če je mogoče, izvajamo naloge večih objektov vzporedno. Dokler se naloge vseh objektov izvajajo na enem samem računalniku to ne vpliva na hitrost izvajanja aplikacije, s primerno porazdelitvijo objektov, ki izvajajo naloge vzporedno, po več računalnikih pa lahko dosežemo bistveno pospešitev aplikacije in večjo izkoriščenost vseh računalnikov, saj so ti večino časa neizkoriščeni.

Današnja računalniška omrežja so tipično heterogena, sestavljena iz računalnikov z različno strojno opremo in operacijskimi sistemi, zato se komunikacija med objekti na različnih računalnikih in primerno porazdeljevanje objektov izkažeta za zapletena problema, ki zahtevata od programerja veliko dela. Pokaže se potreba po sistemu, ki bi poskrbel za navedeni težavi in ju skril pred programerjem, tako da bi se ta lahko v kar največji meri posvetil sami aplikaciji.

V tem delu bomo najprej predstavili pojem objektno usmerjene aplikacije in opisali objektni model CORBA, ki rešuje problem komunikacije med objekti ne glede na njihovo izvedbo in lokacijo na omrežju. Nadaljevali bomo s splošnim pregledom problema porazdeljevanja bremena in ga umestili v CORBA okolje.

Objektni model je zelo splošen in se ne ukvarja s problemom porazdeljevanja, pa tudi v izvedbah CORBA specifikacije, ki so danes na voljo, takšnega sistema ne zasledimo, zato bomo v nadaljevanju predstavili svoj sistem za avtomatsko porazdeljevanje bremena v CORBA okolju.

Od naše izvedbe sistema za porazdeljevanje bremena bomo zahtevali naslednje lastnosti:

- sistem naj omogoča porazdeljevanje med računalniki na poljubnem omrežju, katerega konfiguracije ni potrebno poznati med prevajanjem, temveč šele ob izvajanju aplikacije

- čim večji del porazdeljevanja bremena naj se opravi brez vednosti odjemalca, ki sistem uporablja

- interakcije med sistemom in odjemalcem naj bodo minimalne, da ne bodo po nepotrebnem obremenjevale omrežnih povezav med računalniki
- delo, ki ga opravlja sistem, naj bo prav tako čim manjše
- sistem naj se vklopi v uporabljeno CORBA okolje brez popravkov njegove izvorne kode
- mogoča naj bo uporaba že obstoječih izvedb objektov. Samim izvedbam objektov zato ni potrebno vedeti za sistem za porazdeljevanje bremena, razen če ga želijo tudi same uporabljati v vlogi odjemalcev uslug drugih objektov
- sistem naj omogoča uporabo različnih porazdeljevalnih strategij. Odjemalci lahko, če želijo, uporabijo lastne porazdeljevalne strategije brez posegov v izvorno kodo sistema
- odjemalci naj imajo možnost nadzora nad porazdelitvijo objektov. Če je to potrebno, naj bo odjemalcu dovoljeno določiti, kje se bo nahajal nek objekt

V poglavju 5 bomo opisali izbrano izvedbo CORBA okolja, arhitekturo našega sistema in vmesnike, ki jih ta kaže programerju. Nadaljevali bomo z opisom njegove izvedbe ter izvedbo štirih porazdeljevalnih strategij. Poglavje bomo zaključili z opisom knjižnice, ki olajša uporabo sistema v jeziku C, in primerom preproste aplikacije.

Sistem bomo preizkusili na dveh različnih omrežjih na problemu iskanja mrežne topologije v mreži vozlišč velikosti $N \times M$ stopnje D, ki ima najmanjšo vsoto maksimalne in povprečne razdalje med vsemi pari vozlišč. Primerjali bomo učinkovitost vseh izvedenih porazdeljevalnih strategij ter pospešitev aplikacije v odvisnosti od števila uporabljenih računalnikov. V zaključku bomo povzeli rezultate in predstavili nekaj smeri, v katerih bi veljalo nadaljevati delo.

2 Objektno usmerjeno programiranje

V poglavju opišemo paradigmo objektno usmerjenega programiranja ter enovite in porazdeljene objektno usmerjene aplikacije. Nadaljujemo z analizo težav pri razvoju porazdeljenih aplikacij in predstavitvijo pojma objektnega modela ter naštejemo najpogosteje uporabljane objektne modele.

2.1 Objektno usmerjene aplikacije

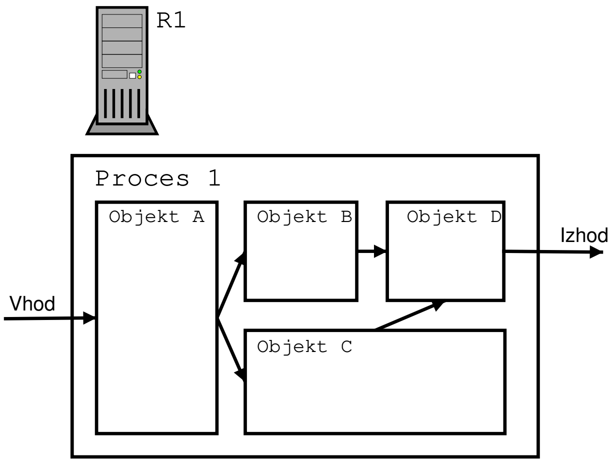
Po paradigmi objektno usmerjenega programiranja lahko aplikacijo predstavimo kot množico objektov, ki komunicirajo med sabo. Vsak od objektov, ki tvorijo aplikacijo, je specializiran za izvajanje točno določenih nalog, katerih primerno zaporedje zagotovi rezultat izvajanja celotne aplikacije.

2.1.1 Enovita objektno usmerjena aplikacija

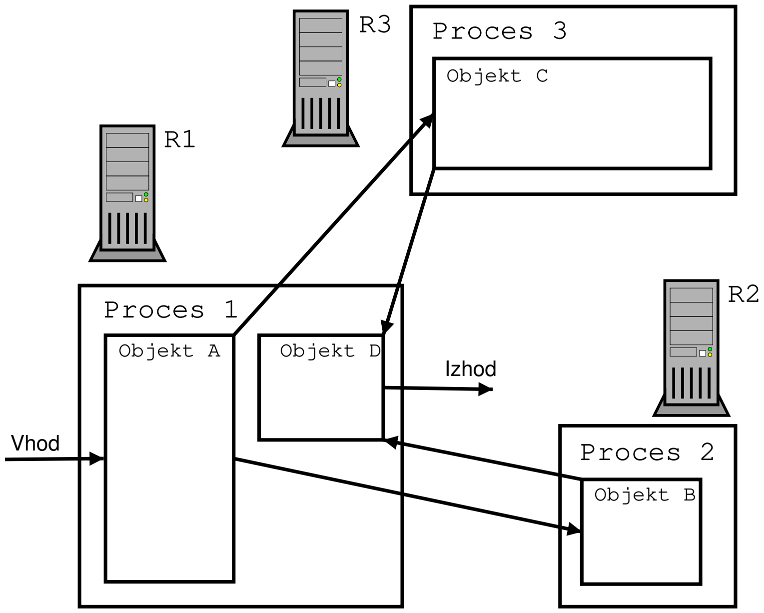
Za objektno usmerjen razvoj tradicionalne enovite aplikacije zadošča objektno usmerjen razmislek in, vendar ne nujno, objektno usmerjen programski jezik, v katerem zapišemo izvorno kodo aplikacije. Postopek prevajanja in povezovanja izvorno kodo prevede in nato poveže v eno samo izvršno datoteko. Izvajanje takšne aplikacije, ki jo prikazuje slika 2.1, tipično poteka v enem samem procesu na enem samem računalniku.

2.1.2 Porazdeljena objektno usmerjena aplikacija

Pogosto je znotraj izvajanja aplikacije mogoče vzporedno izvajati naloge dveh ali več objektov, če so te med seboj neodvisne. Izvajanje nalog posameznih objektov v več procesih na več računalnikih bi torej pospešilo izvajanje aplikacije kot celote in tako povečalo njeno učinkovitost. Takšno razmišljanje nas privede do pojma porazdeljene objektno usmerjene aplikacije, predstavljene na sliki 2.2, za katero je značilno, da se poljuben objekt, ki predstavlja osnovno enoto aplikacije, lahko izvaja v lastnem procesu ter zato tudi na poljubni centralni procesni enoti (CPE) v sistemu.



Slika 2.1: Enovita objektno usmerjena aplikacija



Slika 2.2: Porazdeljena objektno usmerjena aplikacija

Takšna porazdelitev objektov omogoča paralelno izvajanje njihovih nalog. Primer na sliki 2.2 omogoča hkratno izvajanje nalog objektov B in C. Porazdelitev objektov po heterogenem računalniškem okolju pa prinaša množico zapletenih problemov. Osnovne težave, ki jih mora programer rešiti, so:

- komunikacija med objekti: preprosti klici metod ne zadoščajo, saj objekti živijo v različnih procesih in je zatorej potrebno uporabiti vsaj deljeni pomnilnik. Objekti, ki živijo na različnih računalnikih, morajo uporabljati komunikacijo preko omrežja
- razlike v strojni opremi: tipična težava, za katero mora poskrbeti programer, je razlika v shranjevanju podatkov, ki so dolgi več besed. Nekatere CPE uporabljajo pravilo debelejšega, druge pravilo tanjšega konca [8] in za pravilen prenos podatkov med različnima sistemoma je potrebno poskrbeti za ustrezno pretvorbo podatkov
- razlike v sistemski programski opremi: različni računalniki lahko uporabljajo različne operacijske sisteme
- razlike v izvedbi objektov: če objekt živi v lastnem procesu, ga načeloma lahko implementiramo v poljubnem programskem jeziku, ki imajo različna pravila za klic funkcij, prenos parametrov in vračanje rezultatov, razlikujejo pa se tudi v dolžini osnovnih podatkovnih tipov

V primeru, ko skušamo porazdeljeno aplikacijo narediti kar se da splošno (torej uporabno na poljubnem računalniškem omrežju, katerega konfiguracije ne poznamo vnaprej), prinaša reševanje zgornjih težav obilico dela. Čas bi programer koristneje porabil za reševanje prvotnega problema, ki naj bi ga reševala aplikacija. Zategadelj se takšen pristop običajno ne izplača.

Pojavi se potreba po:

- splošnem objektnem modelu , ki bi organizirano predstavil objektne koncepte in terminologijo [1]
- izvedbi objektnega modela , ki bi omogočila aplikacijskemu programerju, da se v čim večji meri posveti razvoju same aplikacije, ne da bi moral skrbeti za “umazane podrobnosti” identifikacije objektov, komunikacije med njimi in pretvorbe podatkov

2.2 Objektni model

Objektni model loči objekte, ki zahtevajo usluge (odjemalce) , od objektov, ki nudijo usluge (strežnikov) , z dobro definiranim vmesnikom. Še prav posebej naj izolira odjemalca od predstavitve podatkov v konkretni izvedbi strežnika in od njegove izvršne kode.

Primera takšnega objektnega modela, ki sta v zadnjem času splošno sprejeta kot standard, sta Microsoftov *Component Object Model* (COM) in objektni model, opisan v specifikaciji *Common Object Request Broker Architecture* (CORBA) [2] skupine Object Management Group (OMG). Na slednjem, zavoljo preprostosti mu recimo kar objektni model CORBA, oziroma na eni njegovih izvedb je osnovan tudi naš sistem za porazdeljevanje objektov.

Prva prepreka za učinkovit razvoj porazdeljenih objektno usmerjenih aplikacij je odpravljena z izbiro primerne izvedbe nekega objektnega modela. Z uvedbo trivialnih paralelizacij - paralelnega izstavljanja med seboj neodvisnih zahtev različnim objektom - bi bilo sedaj mogoče močno pospešiti izvajanje takšnih aplikacij. Seveda pa do takšne pospešitve ne pride - še pogosteje se izvajanje upočasni - če vsi objekti živijo v procesih, ki se izvajajo na enem samem računalniku.

2.3 Porazdelitev objektov

Za čim hitrejše izvajanje aplikacij je potrebno objekte primerno porazdeliti po računalnikih, ki so aplikaciji na voljo. Seveda moramo upoštevati, da ima posamezen računalnik vstavljeno le neko podmnožico izvedb objektov, ki jih aplikacija potrebuje.

Tudi učinkovito porazdeljevanje objektov po računalnikih omrežja, katerega konfiguracije ne poznamo vnaprej, je zahtevna naloga. Razviti moramo postopke za aktivacijo objektov na oddaljenih računalnikih in sestavljanje ter prenos seznamov objektov, ki so na voljo na posameznem računalniku, saj tega objektni model sam ne predpisuje niti ne izvede. Očitna rešitev težav se kaže v vpeljavi sistema za avtomatsko porazdeljevanje objektov, osnovanega na izbranem objektnem modelu, ki ga bomo opisali v prihodnjih poglavjih.

3 CORBA

V tem poglavju predstavimo objektni model, opisan v specifikaciji CORBA, uveljavljen standard, ki ga je razvila skupina Object Management Group (OMG). Navedemo kratek seznam pomembnejših izrazov, nujnih za razumevanje CORBE, in podrobneje opišemo nekatere osnovne elemente, predvsem pa pojme, ki imajo poseben pomen za vzporedno programiranje in izvedbo našega sistema za porazdeljevanje objektov.

3.1 Object Management Architecture

Skupina OMG je bila ustanovljena leta 1989, da bi rešila težave razvoja prenosljivih porazdeljenih aplikacij za heterogene računalniške sisteme. Od takrat je bila deležna izjemne podpore ključnih podjetij v računalniški industriji in je danes največji konzorcij proizvajalcev programske opreme z več kot 800 člani. Za slednje se lahko zahvali predvsem določitvi primernih visokonivojskih abstrakcij, ki med razvojem aplikacij skrijejo nizkonivojske podrobnosti [3]. Predvsem prva specifikacija - Object Management Architecture (OMA) - in njeno jedro, specifikacija Common Object Request Broker Architecture (CORBA), predstavljata arhitekturno osnovo, ki je tako dovolj bogata kot tudi dovolj prožna, da zadošča za razvoj širokega nabora porazdeljenih sistemov.

OMA uporablja dva povezana modela, da opiše pojem porazdeljenih objektov in njihovih medsebojnih odnosov na način, neodvisen od uporabljene strojne in sistemske programske opreme ter objektne tehnologije. *Objektni model* definira, kako opisati vmesnike objektov, porazdeljenih po heterogenem okolju, *referenčni model* pa opisuje razmerja med objekti in nekatere vnaprej določene kategorije objektov. Za naše delo je zanimiv predvsem *objektni model*.

3.2 Objektni model CORBA

CORBA predstavlja nabor od platforme neodvisnih programskih vmesnikov in modelov za porazdeljene objektno usmerjene aplikacije. Zavoljo njene neodvisnosti od programskih jezikov, strojne opreme, operacijskega sistema in mrežnih protokolov je primerna za učinkovit razvoj novih aplikacij in njihovo integracijo v obstoječe porazdeljene sisteme.

3.2.1 Terminologija

Podobno kot vse obsežne tehnologije ima tudi CORBA svoje lastno izrazje, katerega obvladovanje je nujno za njeno dobro razumevanje. Nekatere najosnovnejše pojme razložimo v naslednjem seznamu:

- *Object Request Broker (ORB)* je osnovni mehanizem, preko katerega objekti za programerja nevidno pošiljajo zahteve drug drugemu in prejemajo odgovore nanje. Odjemalcu njihovih uslug ni potrebno poznati podrobnosti komunikacije med objekti, izvedbe posameznih objektov in tudi ne lokacije objekta. ORB je torej osnova vsake CORBA aplikacije in osnova medsebojnega sodelovanja aplikacij tako v homogenih kot heterogenih računalniških okoljih [1]

- *CORBA objekt* je navidezna enota, ki jo lahko ORB poišče in ji lahko odjemalci njenih uslug pošiljajo zahteve. Navidezna zato, ker za odjemalca obstaja samo skozi objektov programski vmesnik, dokler ni realiziran v neki konkretni izvedbi, napisani v nekem programskem jeziku. Prav tako navidezni pomnilnik v operacijskem sistemu pravzaprav ne obstoji, temveč je simuliran s fizičnim pomnilnikom

- *Odjemalec* je enota, ki izda zahtevo CORBA objektu. Odjemalec je lahko drug CORBA objekt ali pa navadna, ne nujno objektno usmerjena aplikacija. Odjemalec in objekt lahko živita v popolnoma ločenih naslovnih prostorih, lahko pa si naslovni prostor tudi delita - sta del istega procesa. Izraz *odjemalec* ima pomen le v kontekstu neke zahteve: nekaj je odjemalec le glede na določen objekt. Izvedba nekega objekta je lahko odjemalec uslug drugih objektov

- *Strežnik* je proces, ki realizira enega ali več CORBA objektov. Prav tako kot *odjemalec* ima tudi ta pomen le v kontekstu konkretne zahteve

- *Zahteva* predstavlja odjemalčev klic operacije nekega CORBA objekta. Zahteve tečejo od odjemalca do strežnika, ki realizira ta objekt, ta pa pošlje rezultate odjemalcu

- *Referenca objekta* je "ročaj", s katerim nedvoumno identificiramo, lociramo in naslovimo CORBA objekt. Odjemalci uporabljajo reference, da naslovijo zahteve na primeren objekt, ne morejo pa dostopati do vsebine referenc ali jih spreminjati. Za odjemalca je referenca nedeljiva enota [3]

3.2.2 Objektni model

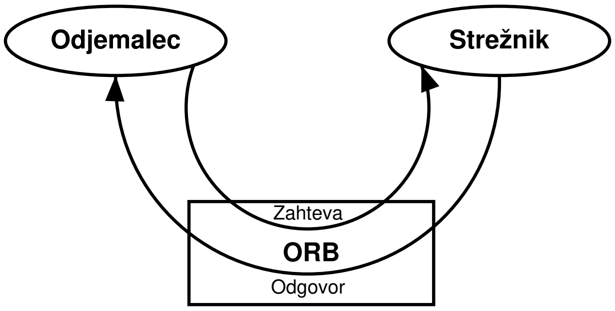
Objektni model najprej opiše koncepte, ki imajo pomen za odjemalca, vključno z ustvarjanjem objektov in njihovo identiteto, zahtevami (angl. requests) in operacijami (angl. operations) ter podatkovnimi tipi in podpisi (angl. signatures) operacij.

Tu je objektni model zelo določen in strog. Sledi opis konceptov, ki se nanašajo na izvedbe objektov: opis metod, njihovega izvrševanja in aktivacije objektov. Pri tem opisu objektni model bolj namiguje na primerne rešitve z namenom ohraniti kar se da veliko svobodo pri izbiri objektne tehnologije, ki bo temelj za izvedbe objektov.

Nekatere značilnosti objektnih sistemov so izven domene objektnega modela in jih ta prepušča izbiri objektne tehnologije. Objektni model ne določa arhitekture same aplikacije, ne obravnava zlepkov (angl. compound) objektov, njihovega razmnoževanja ipd. Prav tako so izven območja objektnega modela podrobnosti programskega toka: ne ukvarja se s tem, ali odjemalci in strežniki tečejo v enem ali več procesih oziroma nitih, in ne določa, kako so procesi ali niti ustvarjeni, uničeni ali sinhronizirani [1].

3.3 Object Request Broker

Osnovno vlogo ORBa kot posrednika med odjemalcem in izvedbo objekta, strežnikom, prikazuje slika 3.1.



Slika 3.1: Pošiljanje zahteve skozi ORB

Odjemalec je entiteta, ki zahteva od objekta izvedbo neke operacije, izvedba objekta pa predstavlja kodo in podatke, ki dejansko realizirajo ta objekt. Naloga ORBa je poiskati izvedbo objekta, na katerega je naslovljena zahteva, pripraviti izvedbo na sprejetje zahteve in posredovati podatke, ki tvorijo zahtevo in njene parametre. Podatki, ki določajo zahtevo, in parametri zahteve se prenašajo v standardizirani obliki, ki jo ORB prevede v predstavitev, primerno platformi, na kateri tečeta odjemalec oziroma strežnik, in jeziku, v katerem sta izvedena.

3.4 Odjemalec

Odjemalec ima dostop do reference nekega objekta, ki jo uporablja, da proži (angl. invokes) operacije: objektu izstavlja zahteve za izvajanje določenih operacij. Odjemalec pozna le logično strukturo objekta (skozi vmesnik, ki ga objekt kaže svoji okolici) in izkuša obnašanje objekta s proženjem njegovih operacij.

Vmesnik, kot ga vidi odjemalec, je popolnoma neodvisen od tega, kje se izvedba objekta nahaja in v katerem programskem jeziku je bila napisana.

Odjemalci vidijo ORB in vmesnike objektov iz perspektive programskega jezika, v katerem so pisani, kar dviguje ORB na programerjev nivo. Odjemalci so popolnoma prenosljivi in bi morali delovati brez sprememb v izvorni kodi s poljubnim ORBom, ki podpira želeni programski jezik, ter s poljubno izvedbo objekta.

3.5 Izvedbe objektov

Izvedba objekta poskrbi za semantiko objekta, ponavadi tako, da definira podatke za instanco objekta in kodo za metode objekta. Pogosto izvedba objekta uporablja dodatno programsko opremo ali usluge drugih objektov, da izvrši želene naloge.

CORBA podpira široko paleto možnosti za izvedbe objektov, ki vključujejo ločene procese, dinamične knjižnice, izvajanje programa za vsako metodo, objektno usmerjeno podatkovno bazo ipd. Načeloma je mogoče podpreti praktično vsako obliko izvedbe objekta.

3.6 Jezik za definicijo programskega vmesnika

CORBA ohrani neodvisnost od programskega jezika tako, da zahteva opis vmesnika v posebnem jeziku Interface Definition Language (IDL) . Pred pisanjem odjemalca ali izvedbe objekta definicijo vmesnikov iz IDLa prevedemo v izbran programski jezik.

Poleg definicije vmesnikov omogoča IDL tudi definiranje različnih sestavljenih podatkovnih tipov (struktur in unij), oštevilčenih vrednosti (angl. enums) ipd., ki jih lahko uporabljamo kot parametre za operacije.

Prevajalnik prevede IDL opis v tri skupine datotek, ki jih potrebujemo za uporabo CORBE v nekem programskem jeziku. V prvo skupino spada definicija vmesnika v izbranem programskem jeziku (definicijska datoteka v jeziku C/C++). V drugo skupino spada koda, ki jo potrebuje odjemalec. Ta omogoča proženje operacij objekta v obliki, ki je naravna v izbranem jeziku, npr. v obliki funkcijskih klicev v jeziku C oziroma klica metod objekta v jeziku C++. Tem funkcijam, ki pretvorijo parametre v standardno obliko in jih skupaj z informacijo o operaciji, ki jo odjemalec želi sprožiti, posredujejo preko ORBa, pravimo *štrclji (angl. stubs)* . Tretja skupina vsebuje kodo, ki jo potrebuje izvedba objekta, da po navodilih ORBa sproži želeno operacijo: *ogrodja (angl. skeletons)* so funkcije, ki jih pokliče ORB, te pa pretvorijo parametre v izvedbi objekta primerno obliko (odvisno od programskega jezika, v katerem je bila izvedba napisana, in strojne opreme, na kateri teče, glej razdelek 2.1.2) ter pokličejo funkcijo, ki dejansko izvede operacijo.

Po prevajanju IDLja v izbrani programski jezik je za izvedbo objekta dovolj napisati le kodo, ki definira želeno semantiko operacij, uporaba objekta v odjemalcih pa je reducirana na npr. preprost klic funkcije.

3.6.1 Primer opisa vmesnika v jeziku IDL

Naslednji primer opiše preprost vmesnik Kalkulator, ki omogoča seštevanje in odštevanje dveh števil.

```
module Primer
{
        interface Kalkulator
        {
                long sestej(in long s1, in long s2);
                long odstej(in long s1, in long s2);
        };
};
```

3.7 Izstavljanje zahtev

Odjemalec uporablja usluge objekta tako, da preko ORBa izstavlja zahteve za proženje operacij nekemu objektu. CORBA pozna dva načina izstavljanja zahtev: *statičnega* in *dinamičnega*.

3.7.1 Statično izstavljanje zahtev

Statično izstavljanje zahtev poteka v obliki, ki je naravna jeziku, v katerem pišemo izvorno kodo odjemalca. V jeziku C je proženje operacije `sestej` v vmesniku `Kalkulator` iz modula `Primer` videti kot običajen klic funkcije:

```
tri = Primer_Kalkulator_sestej(kalkulator, 1, 2, env);
```

Parametri takšnega klica so:

1. `kalkulator`: referenca objekta, katerega operacijo želimo sprožiti
2. `1, 2`: dejanska parametra operacije, seštevanca
3. `env`: struktura, preko katere lahko npr. izvemo, kako je klic uspel: vsebuje morebitne izjeme, ki jih sintaksa jezika C ne podpira. V primeru objektno usmerjenega jezika, ki sintaktično podpira izjeme, bi ta ne bil potreben

Funkcijo (štrcelj) `Primer_Kalkulator_sestej` generira IDL prevajalnik, njena naloga pa je, da parametre prevede v obliko, primerno za posredovanje preko ORBa, in jo skupaj s podatki, ki identificirajo metodo in objekt, na katerega se nanaša referenca `kalkulator`, pošlje ORBu.

Statično izstavljanje zahtev je za programerja zelo naraven način uporabe CORBA objektov. Zahteva, da vmesnik objektov, ki jih uporabljamo, poznamo v času prevajanja (potrebujemo IDL opis vmesnikov). Uporabo statične izstavitve zahteve pri vzporednem programiranju onemogoča to, da se takšna zahteva obnaša ravno tako kot običajen klic funkcije - blokira izvajanje glavnega programa, dokler zahteva ne vrne rezultata.

3.7.2 Dinamično izstavljanje zahtev

Dinamično izstavljanje zahtev naj bi se uporabljalo predvsem, kadar vmesnik ni znan med prevajanjem. Odjemalec lahko namreč med izvajanjem iz *skladišča vmesnikov (Interface Repository, IR)* izve za vmesnik poljubnega objekta, imena njegovih operacij, imena in tipe njihovih parametrov ter tip rezultata. Med izvajanjem dinamično sestavi zahtevo ter jo, tako kot v *statičnem* primeru, pošlje objektu s posredovanjem ORBa.

Takšen način uporabe objektov je precej bolj neroden za uporabo, saj moramo pri pošiljanju zahteve navesti tipe parametrov in njihove vrednosti. Navajamo primer *dinamične* izstavitve zahteve po proženju operacije `sestej`, spet v jeziku C:

```
CORBA_long a = 1, b = 2;
CORBA_long tri;

CORBA_ORB_create_list(orb, 2, &parameter_list, env);
CORBA_NVList_add_item(parameter_list,
                      "s1", TC_long, &a, sizeof(a),
                      0, env);
CORBA_NVList_add_item(parameter_list,
                      "s2", TC_long, &b, sizeof(b),
                      0, env);
result = malloc(sizeof(CORBA_NamedValue));
result->argument._type = TC_long;
result->arg_modes = CORBA_ARG_OUT;
result->len = sizeof(tri);
result->argument._value = &tri;

CORBA_Object_create_request(kalkulator, context,
                            "sestej", parameter_list, result,
                            &request, request_flags, env);
CORBA_Request_send(request, send_flags, env);
CORBA_Request_get_response(request, response_flags, env);
```

Poleg parametrov `kalkulator` in `env`, ki ju že poznamo, se tu pojavljajo še naslednji:

1. `parameter_list`: seznam parametov
2. `result`: hramba za rezultat
3. `request`: struktura, ki povezuje vse potrebne podatke za izstavitev zahteve

Ostale parametre bomo zavoljo preprostosti prezrli.

Običajno tudi klic `CORBA_Request_get_response` blokira izvajanje glavnega programa, dokler odjemalec ne dobi rezultata. Takšen način dinamičnega izstavljanja zahtev je enakovreden statičnemu. Klic funkcije `CORBA_Request_get_response` pa takoj, četudi rezultat operacije še ni na voljo, vrne kontrolo glavnemu programu, če v parametru `response_flags` postavimo zastavico `CORBA_RESP_NO_WAIT`. Z uporabo takšnega načina proženja operacij, s ti. zakasnjenimi sinhronimi operacijami (angl. deferred synchronous operations) [2], lahko zahtevamo proženje neke operacije in med čakanjem na rezultat počnemo druge stvari, kar je primerno za vzporedno programiranje[1].

3.8 Primer izvedbe objekta

Kodo, potrebno za izvedbo odjemalca, smo si ogledali že med razpravo o izstavljanju zahtev, za lažje razumevanje zgoraj navedenega pa si oglejmo še izvedbo objekta z vmesnikom Kalkulator v jeziku C. Uporabljen je bil ORB ORBit 0.5.6.

Objekt bomo izvedli kot samostojen proces. Poleg kode, ki jo generira IDL prevajalnik, moramo izvesti še dejanski metodi `sestej` in `odstej`:

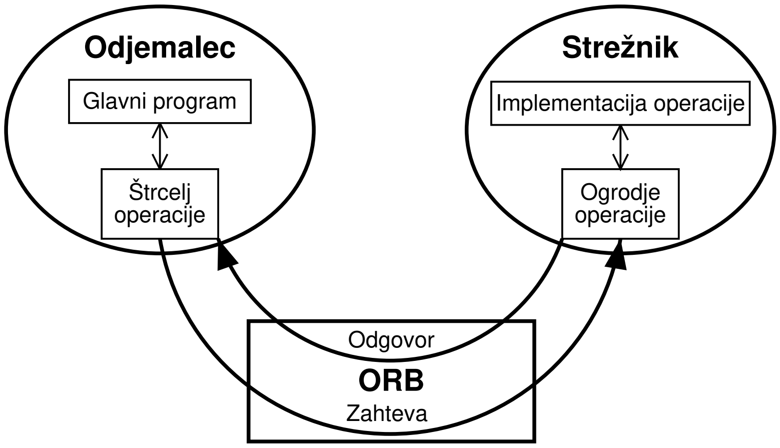
```
static CORBA_long
impl_Primer_Kalkulator_sestej(impl_POA_Primer_Kalkulator *servant,
                              CORBA_long s1, CORBA_long s2,
                              CORBA_Environment *ev)
{
        return s1 + s2;
}

static CORBA_long
impl_Primer_Kalkulator_odstej(impl_POA_Primer_Kalkulator *servant,
                              CORBA_long s1, CORBA_long s2,
                              CORBA_Environment *ev)
{
        return s1 - s2;
}
```

Ostane nam le še, da v kodo, ki jo je generiral IDL prevajalnik, vnesemo kazalce na zgornji dve funkciji in aktiviramo objekt, kar običajno, če je edini namen našega programa zgolj izvedba objekta, opravimo kar v funkciji `main`. Vso ostalo kodo lahko na osnovi vmesnika in izbranega jezika generira IDL prevajalnik.

Slika 3.2 prikazuje statično proženje operacije. Zavedati se moramo, da slikama 3.1 in 3.2 navkljub ni nujno, da ORB sploh obstaja kot poseben in od odjemalca ter strežnika ločen posrednik. Lahko in pogosto tudi je izveden kot del odjemalca in strežnika, npr. kot dinamično povezljiva knjižnica. Uporabo različnih izvedb ORBov pri odjemalcu in strežniku pa omogoča standardiziran protokol komunikacije *General Inter-Orb Protocol, GIOP* [2] .

[1] Asynchronous Method Invocation (AMI) , del specifikacije CORBA 3.0, naj bi olajšal neblokirajoče proženje operacij.



Slika 3.2: Podrobnosti statičnega pošiljanja zahteve skozi ORB

3.9 Skladišče izvedb

Zgoraj opisane značilnosti objektnega modela CORBA zadovoljivo rešujejo večino težav porazdeljenih objektnih sistemov, ostaja pa še vprašanje, kako odjemalec pridobi referenco do izvedbe nekega objekta.

Referenco je vedno mogoče zapisati v obliki znakovnega niza. To lahko stori strežnik, ko ustvari nov primerek objekta, in jo zapiše v datoteko, od koder jo lahko preberejo odjemalci. Ta način sicer omogoča odjemalcem, da izvejo za referenco objekta, za katerega skrbi že pognan strežnik, vendar je neprimeren za uporabo ter ne rešuje težave samodejnega poganjanja želenih strežnikov. Tu nastopi *skladišče izvedb (Implementation Repository, IMR)*, ki skrbi za posredovanje referenc objektov odjemalcem, rešuje pa tudi vrsto drugih težav.

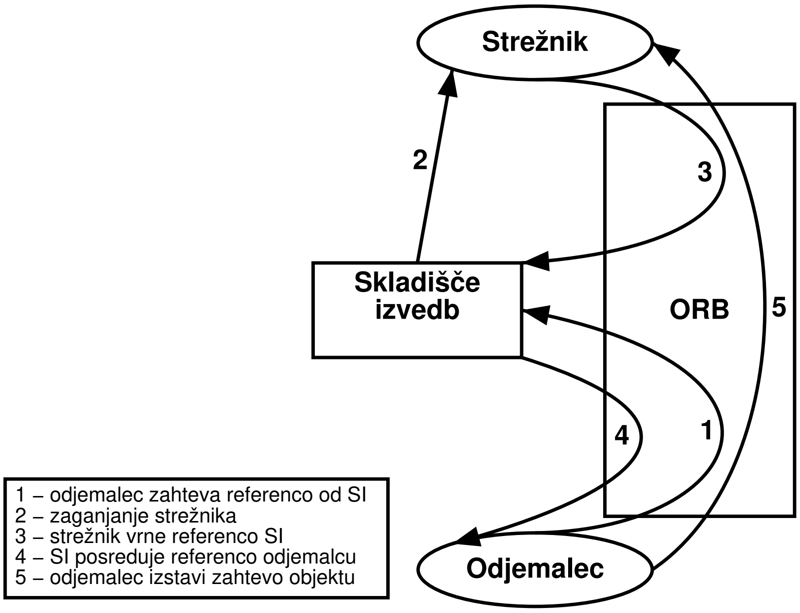
Specifikacija CORBA ne podaja natančnega opisa skladišča izvedb, temveč le svetuje, kakšne lastnosti naj bi imelo. Pomanjkanje standardizacije je namerno, saj so skladišča izvedb močno odvisna od strojne in sistemske programske opreme. Skrbijo lahko tudi za učinkovitost sistema, porazdeljevanje bremena, selitve objektov ipd. ter so zato mesto, kjer se lahko posamezne izvedbe ORBov prilagodijo ciljnemu okolju.

V splošnem pa velja, da mora skladišče izvedb:

- hraniti seznam vseh znanih vmesnikov ter informacijo o tem, kako pognati strežnik, ki vsebuje izvedbo vmesnika
- hraniti seznam referenc ustvarjenih objektov in jih posredovati odjemalcem
- na zahtevo pognati strežnik, ki vsebuje izvedbo želenega vmesnika

Odjemalec, ki pozna naslov skladišča izvedb (to je pogosto izvedeno kot CORBA objekt, tako da odjemalcu zadošča njegova referenca, ki je običajno znana ob zagonu odjemalca), lahko od njega torej zahteva aktivacijo objekta z določenim vmesnikom, skladišče izvedb pa mu vrne referenco primernega objekta, če ta že obstaja. V nasprotnem primeru poizkusi skladišče izvedb pognati primeren strežnik ter vrniti referenco na tako ustvarjen nov objekt. S tako dobljeno referenco lahko odjemalec izstavi prvo zahtevo objektu.

Opisano pridobivanje referenc, kjer je skladišče izvedb samostojen CORBA objekt, prikazuje slika 3.3.



Slika 3.3: Pridobivanje reference iz skladišča izvedb

4 Porazdeljevanje bremena

V poglavju predstavimo problem porazdeljevanja bremena in pojem *strategije porazdeljevanja bremena* ter opišemo najpomembnejše tipe le-teh. Opišemo pojem obremenitve in pogoste težave, ki jih morajo porazdeljevalne strategije reševati.

4.1 Osnove

Ena večjih težav porazdeljenih računalniških sistemov in aplikacij, ki tečejo na njih, se kaže v tem, da so nekateri deli sistema prezasedeni, medtem ko drugi ostajajo neizkoriščeni. Za dobro izkoriščenost računskih zmogljivosti je torej potrebno dobro porazdeliti celotno računsko breme, ki ga predstavljajo izračuni v aplikaciji, med vire, ki so v porazdeljenem sistemu na voljo, in tako doseči kar se da enakomerno izkoriščenost vseh delov sistema ter čim hitrejše izvajanje aplikacije.

Problem porazdeljevanja bremena lahko predstavimo kot problem dodeljevanja množice poslov računalnikom, ki jih bodo izvršili, tako da so zadovoljeni določeni ciljni kriteriji. Med najpogostejše ciljne kriterije spadata čim krajši čas izvajanja celotne aplikacije kot unije vseh poslov ali pa čim bolj enakomerna obremenitev vseh delov računalniškega sistema. Pogosto tem glavnim kriterijem dodamo še dodatne omejitve, ki se nanašajo na naravo same aplikacije in okolja, v katerem teče. Med slednje spadajo različne funkcijske (poljubnega posla ne moremo dodeliti kateremu koli računalniku) in časovne (če gre za sisteme v realnem času) omejitve, odpornost na napake v primeru kritičnih aplikacij, varnostne zahteve sistema ipd. [6]

4.2 Strategije porazdeljevanja bremena

Postopek, ki dodeljuje posle računalnikom, kot je bilo opisano zgoraj, imenujemo *strategija porazdeljevanja bremena* .

K izvedbi strategije za porazdeljevanje bremena lahko, v odvisnosti od tega, kakšno količino informacij o sistemu bomo uporabljali uporabili ter kje in kdaj poteka porazdeljevanje, pristopimo na več načinov.

4.2.1 Statične in dinamične strategije

Glede na informacijo o trenutnem stanju računalniškega sistema, ki jo strategije uporabljajo, jih delimo na:

- *Statične* porazdeljevalne strategije, ki ne uporabljajo nobene informacije o trenutnem stanju sistema. Statične strategije lahko uporabimo tako, da že ob zagonu aplikacije porazdelimo vse posle, če jih seveda poznamo. Lahko so deterministične (npr. dodelijo posel P_n računalniku $R_{n\ mod\ m}$, če je v sistemu m računalnikov) ali pa verjetnostne (angl. probabilistic) (npr. dodelijo nek posel računalniku R_n z verjetnostjo p_n)

- *Dinamične* porazdeljevalne strategije, ki se pri odločanju o tem, kateri računalnik bo izvrševal nek posel, opirajo na čim bolj ažurno informacijo o stanju sistema, predvsem o trenutni ali povprečni obremenjenosti računalnikov. Očitno je, da spričo potrebe po občasnem zbiranju podatkov o sistemu dinamične strategije zahtevajo več dela, ki pa ga običajno upravičijo z opazno boljšim zadovoljevanjem ciljnih kriterijev. Posebna oblika dinamičnih strategij so prilagodljivi (adaptive) postopki, ki med samim izvajanjem dinamično spreminjajo svoje parametre, tako da se prilagajajo spremembam v sistemu

4.2.2 Centralizirane in porazdeljene strategije

Strategije so lahko centralizirane ali porazdeljene . Za centralizirane strategije je značilno, da za porazdeljevanje skrbi ena sama komponenta sistema, medtem ko pri porazdeljenih strategijah za porazdeljevanje skrbijo vsi deli sistema. Centralizirano strategijo je laže izvesti, vendar je manj zanesljiva, saj njen izpad popolnoma onemogoči porazdeljevanje bremena in pogosto tudi kakršno koli izvajanje aplikacije. Osrednja komponenta, ki skrbi za porazdeljevanje, lahko pri velikem številu poslov, ki jih je potrebno porazdeliti, postane ozko grlo sistema, kar vodi k slabši učinkovitosti.

4.2.3 Predkupne in nepredkupne strategije

Porazdeljevanje bremena lahko opravljamo le ob pojavu novih poslov - tu govorimo o t.i. nepredkupnih (angl. non-preemptive) postopkih. Postopki, ki dovoljujejo, da že obstoječe in porazdeljene posle selimo med računalniki z namenom izboljšati njihovo porazdelitev, so predkupni (angl. preemptive). Na žalost je takšna selitev običajno zelo zahtevna, saj moramo, če se je posel že začel izvajati, poleg njega samega preseliti tudi njegovo stanje. V splošnem je težko odgovoriti, katere strategije se obnesejo bolje, saj so različne študije pripeljale do različnih odgovorov [4]. Učinkovitost predkupnih strategij je močno odvisna tudi od kompleksnosti stanja dodeljevanih poslov, zato se za primerno vrsto strategije najlaže odločimo na primeru konkretne aplikacije.

4.3 Težave porazdeljevalnih postopkov

Pri porazdeljevalnih postopkih se pogosto pojavljajo problemi, ki morajo biti ustrezno rešeni. V splošnem velja, da se zapletenejše strategije ubadajo z večjim naborom težav.

4.3.1 Delo potrebno za porazdeljevanje bremena

Tudi sam porazdeljevalni postopek predstavlja breme za računalniški sistem. Zbiranje podatkov o sistemu in morebitne selitve poslov obremenjujejo tako računske kot komunikacijske zmogljivosti sistema. Paziti moramo, da to breme ne postane preveliko v primerjavi z bremenom, ki ga predstavlja izvajanje aplikacije, saj tudi najboljša porazdeljevalna strategija ne upraviči svoje uporabe, če porazdeljevanje porabi preveč časa. Tipičen primer so optimalne porazdeljevalne strategije, katerih časovna zahtevnost je eksponentna, saj rešujejo NP-polne probleme.

4.3.2 Zastarele informacije o stanju sistema

S porazdeljevanjem bremen se stanje sistema nepredvidljivo spreminja. Informacije o stanju sistema, ki jih uporablja porazdeljevalni postopek, zato hitro zastarajo, odločitve na osnovi netočnih, zastarelih informacij pa vodijo do neoptimalnega ali celo popolnoma napačnega porazdeljevanja. Prepogosto zbiranje informacij o sistemu predstavlja dodatno delo, ki se spet manifestira v slabših rezultatih porazdeljevalnega postopka. Informacije o stanju sistema zato še vedno ne velja zbirati prepogosto, pomagajo pa hevristični popravki te informacije na osnovi preselitev in na novo porazdeljenih poslov.

4.3.3 Preobremenjevanje računalnikov

Računalnikom z nizko obremenitvijo je posel dodeljen pogosteje. Predvsem pri porazdeljenih strategijah, kjer posamezni deli sistema ne vedo za odločitve drugih delov, se pogosto zgodi, da se računalniki z nizko obremenitvijo hitro preobremenijo, saj se jim naenkrat dodeli več poslov.

4.3.4 Osciliranje

Do osciliranja pride, ko porazdeljevalna strategija vztrajno sprejema zaporedje nasprotujočih si odločitev. Predkupna porazdeljevalna strategija lahko zaradi kratkotrajne preobremenitve enega računalnika (zaradi npr. redne naloge, ki jo računalnik opravlja vsakih nekaj minut) preseli posle na drug računalnik in takoj nato spet nazaj, ker je prvotni računalnik zdaj neizkoriščen.

4.3.5 Premetavanje

Premetavanje je običajno posledica visoke obremenjenosti sistema. Takrat več preobremenjenih računalnikov poskuša preseliti svoje posle na druge računalnike. Ti postanejo preobremenjeni in poskušajo preseliti posle spet drugam, kar se ponavlja v neskončnost. Sistem ne opravlja več koristnega dela, temveč vse svoje zmogljivosti porabi za selitve poslov med računalniki.

4.4 Kazalci obremenitve

Za uspešno izvedbo dinamičnih porazdeljevalnih strategij je potrebna predvsem čim boljša informacija o stanju sistema, v njenem okviru pa je še posebej pomembna obremenitev posameznih računalnikov. Kvantitativno jo predstavimo s *kazalcem obremenitve* .

V literaturi je obravnavanih veliko takšnih kazalcev [5]. Tipični primeri so:

- *število procesov v čakalni vrsti CPE* računalnika
- *izkoriščenost* računalnika: delež časa v zadnjih n sekundah, ko je bila CPE zaposlena
- *odzivni čas* računalnika

Izkaže se, da so preprosti kazalci učinkovitejši od sestavljenih (npr. linearne kombinacije več preprostih kazalcev). Ravno tako dosežemo boljše rezultate s povprečenjem vrednosti preko krajšega intervala, vendar velja biti tu previden, saj predolgi intervali povzročijo neobčutljivost kazalca za krajše spremembe v obremenitvi.

4.5 Integracija porazdeljevanja v CORBA okolje

Računalniško okolje, v katerem tečejo CORBA aplikacije, je običajno zelo raznoliko. Zaradi tega je splošen model porazdeljevanja bremena v CORBA okolju neprimerno bolj zapleten kot tradicionalni pristopi k porazdeljevanju bremena, ki običajno temeljijo na predpostavki čim večje enovitosti tako množice poslov kot množice računalnikov. Tradicionalni pristopi običajno predpostavljajo, da lahko poljuben posel dodelimo poljubnemu računalniku. Rahlo bolj splošen, a za naš namen še vedno preveč omejujoč model sestavljajo razredi poslov in računalnikov ter pravila, ki določajo, v kateri razred naj spada posel, da ga lahko dodelimo računalniku iz danega razreda.

Model za porazdeljevanje bremena v CORBA okolju mora biti zato dovolj splošen za uspešno uporabo ne glede na programski model, programski jezik, način izvedbe objektov in strojno opremo, ki bo tvorila okolje, v katerem bo tekla aplikacija, saj CORBA ne predpisuje ničesar od navedenega.

Splošen model obravnava porazdeljevanje bremena v CORBA okolju na treh nivojih:

- na *nivoju objektov* poteka porazdeljevanje bremena predvsem preko izbire referenc. Tipično odjemalec pove, s kakšnim objektom želi delati, porazdeljevalna strategija pa mu dodeli referenco najprimernejšega objekta. Vsem nadaljnjim zahtevam, ki bodo izstavljene tej referenci, bo postregel isti primerek objekta
- *nivo izvedb objektov* temelji na dejstvu, da specifikacija CORBA ne določa, kako so strežniki izvedeni. Ponuja se npr. možnost, da strežnik izvedemo kot množico procesov z enako funkcionalnostjo, razdeljenih po množici računalnikov. Strežnik izstavljeni zahtevi ugodi tako, da zahtevano računanje opravi v procesu na najmanj obremenjenem računalniku
- *sistemski nivo* CORBA aplikacije sestoji iz virov, ki jih aplikacija potrebuje, in računalnikov. Vir ni nujno fizično sredstvo, temveč je lahko npr. določen proces ali baza podatkov. Vir je enota, ki je sposobna izpolnjevanja njej izstavljenih zahtev. Na sistemskem nivoju lahko npr. uporabimo tradicionalno prirejanje procesov računalnikom

Podrobnejši opis te razvrstitve si lahko bralec ogleda v [6], od izvedbe konkretnega postopka porazdeljevanja pa je odvisno, kolikšen del navedenih mehanizmov bo ta izkoriščal.

5 Sistem za porazdeljevanje bremena ODF

V tem poglavju predstavimo uporabljeno izvedbo CORBA okolja ter razložimo arhitekturo, delovanje in nekatere podrobnosti izvedbe našega sistema za porazdeljevanje bremena. Zaključimo z opisom štirih izvedenih porazdeljevalnih strategij.

5.1 CORBA okolje

Za izvedbo našega sistema za porazdeljevanje bremena, imenujmo ga *Object Distribution Framework, ODF*, moramo najprej izbrati primerno CORBA okolje. Od izbrane izvedbe bomo zahtevali predvsem, da naj teče na čim širšem naboru strojne opreme in operacijskih sistemov, saj bo to omogočalo njegovo uporabo na heterogenih omrežjih. Zaželeni lastnosti sta tudi čim večja hitrost in varčnost s spominom, da bi že samo okolje sistema ne obremenjevalo preveč.

Uporabili smo izvedbo CORBA specifikacije *ORBit* [1] in njemu pripadajoče skladišče izvedb *Object Activation Framework, OAF*[2].

Tako ORBit kot OAF sta bila razvita za potrebe projekta GNOME [13] in tečeta na vseh modernih UNIX sistemih ter na vseh danes razširjenih procesorskih arhitekturah. Odlikuje ju predvsem velika hitrost, k njuni izbiri pa je pripomoglo tudi to, da je bila na voljo izvorna koda obeh, ter da sem sam sodeloval pri njunem razvoju in zato dobro poznam njuno delovanje, prednosti in slabosti.

5.1.1 ORBit

ORBit je izvedba različice 2.2 specifikacije CORBA. Izvedena je v obliki dinamične knjižnice, zaradi intenzivne uporabe v namiznem okolju GNOME pa se odlikuje predvsem z izredno hitrostjo. Teče tako na UNIX kot na Windows 95/98/NT/2000 operacijskih sistemih.

Pripadajoči IDL prevajalnik prevaja IDL v jezik C, dodatno pa je podprtih veliko modernih jezikov [14]. Od zanimivejših omenimo C++, Pascal, Perl in Haskell.

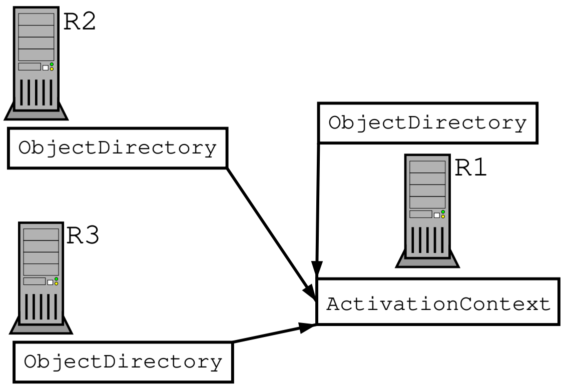
[1] Med razvojem sistema smo uporabljali različice od 0.5.0 do 0.5.5; zadnja različica v času pisanja tega dela je 0.5.7.

[2] Od tod tudi ime našega sistema, ODF.

ORBit podpira večino CORBA 2.2 specifikacije, vključno z dinamičnim izstavljanjem zahtev, kar je za vzporedno programiranje še posebej pomembno. Različica 0.4.3, s katero sem začel delo, ni podpirala odloženega proženja zahtev, ki je prav tako potrebno za vzporedno programiranje, zato sem ga izvedel sam. Te spremembe so vključene v različico 0.5.0 in kasnejše.

5.1.2 Object Activation Framework

OAF je skladišče izvedb, narejeno posebej za uporabo z ORBitom. Podpira izvedbe objektov v obliki dinamičnih knjižnic, samostojnih procesov in tovarn objektov[3] (angl. object factories). OAF, njegovo arhitekturo prikazuje slika 5.1, tvorita dva vmesnika: prvi je ObjectDirectory (OD), ki hrani seznam izvedb objektov, nameščenih na računalniku, kjer teče, ter skrbi za njihovo poganjanje. Drugi, ActivationContext (AC), pa vzdržuje seznam ODjev za množico računalnikov ter omogoča izbiro objekta z lastnostmi, ki jih poda programer. Oba objekta sta izvedena v okviru enega samega strežnega procesa `oafd` .



Slika 5.1: Arhitektura OAF

Izvedbo objekta namestimo tako, da na predvideno mesto v datotečni strukturi postavimo datoteko , ki opisuje izvedbo. Datoteka vsebuje seznam lastnosti objekta v obliki parov $(ime, vrednost)$, vrsto izvedbe objekta, niz, ki opisuje izvedbo, in mesto, kamor je izvedba nameščena. Navedli bomo primer takšne datoteke za izvedbo vmesnika `Kalkulator` iz zgornjega primera:

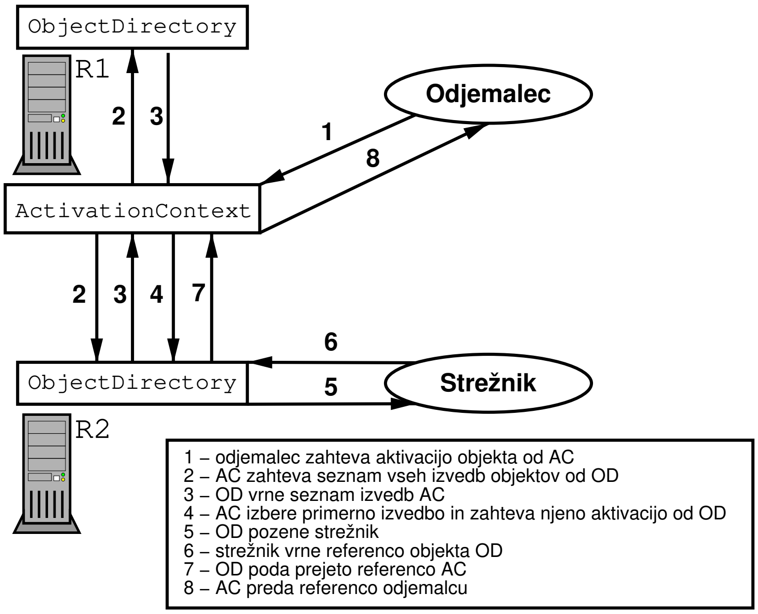
```
<oaf_server iid="OAFIID:Primer_Kalkulator_Izvedba1" type="exe" location="">
        <oaf_attribute name="repo_ids" type="stringv">
                <item value="IDL:Primer/Kalkulator:1.0"/>
        </oaf_attribute>
        <oaf_attribute name="barva" type="string" value="zelena"/>
</oaf_server>
```

[3]Tovarne objektov so procesi, ki skrbijo za več primerkov objektov, lahko tudi različnih razredov.

Programer izbere nek objekt tako, da v posebnem povpraševalnem jeziku (*OAF query language*) navede lastnosti želenega objekta ter nekatere druge omejitve. Objekt, ki realizira vmesnik `Kalkulator` z izvedbo, ki naj teče na računalniku `machiavelli`, izberemo takole:

```
requirements = "repo_ids.has('IDL:Primer/Kalkulator:1.0') AND " \
               "hostname == 'machiavelli'";
kalkulator = AC::activate(requirements, order, flags, context);
```

AC izbere podmnožico izvedb, ki ustrezajo omejitvam, podanim s parametrom `requirements`, ter zahteva od OD na računalniku, kjer je nameščena ustrezna izvedba, referenco objekta, ki ga ta izvedba realizira. OD po potrebi požene primerno izvedbo ter vrne referenco. Ta postopek *aktivacije* objekta (ki ni nič drugega kot pridobivanje reference objekta) prikazuje slika 5.2. Za primerjavo si velja ogledati tudi splošni postopek pridobivanja reference objekta iz skladišča izvedb na sliki 3.3. Parameter `order` je namenjen urejanju vrstnega reda vseh ustreznih izvedb, s parametrom `flags` pa lahko zahtevamo, da je referenca namenjena izključno našemu odjemalcu, da se vedno zažene nova izvedba ipd. Parameter `context` je referenca objekta iz razreda `CORBA::Context`, ki vsebuje podatke o okolju, v katerem teče aplikacija.



Slika 5.2: Aktivacija objekta s pomočjo OAF

AC omogoča tudi povpraševanje po množici strežnikov, ki ustrezajo navedenim omejitvam. To opravimo z metodo `query` :

```
server_info_list = OAF::ActivationContext::query(requirements, order);
```

Pri pisanju programov, ki uporabljajo OAF, lahko tega uporabljamo na nivoju CORBE, lahko pa uporabimo knjižnico `liboaf` , ki olajša uporabo v jeziku C. Poleg lažje uporabe OAF programerju dovoljuje tudi, da na svoj način izvede dostop do referenc OD in AC na podanem računalniku[4]. Reference hranijo *registracijska mesta* . Programer lahko definira poljubno število registracijskih mest. Izvedemo jih lahko na poljuben način, programer mora knjižnici le podati kazalce na pet funkcij, ki jih mora opravljati vsako registracijsko mesto:

- `lock` in `unlock`, ki poskrbita za zaklepanje dostopa do registracijskega mesta
- `register` in `unregister`, s katerima OAF doda novo oziroma odstrani obstoječo referenco za OD ali AC na določenem računalniku
- s funkcijo `check` OAF zahteva referenco želenega OD ali AC

V primeru, ko želeni objekt ni registriran, bo OAF poizkusil z enim od *registracijskih aktivatorjev* pognati strežnik `oafd` na ustreznem računalniku ter registrirati tako dobljeni referenci OD in AC na vseh registracijskih mestih. Tudi registracijske aktivatorje lahko definira programer sam.

Knjižnica `liboaf` že vsebuje po eno registracijsko mesto in aktivator, vendar je registracijsko mesto izvedeno z lokalnimi datotekami, aktivator pa je sposoben pognati strežnik le na lokalnem računalniku.

Kot ogrodje sistema za porazdeljevanje bremena bomo lahko uporabili ORBit in OAF, če bomo uporabili registracijsko mesto, ki ne bo vezano na lokalni datotečni sistem, in aktivator, ki bo sposoben pognati strežnik na oddaljenem računalniku.

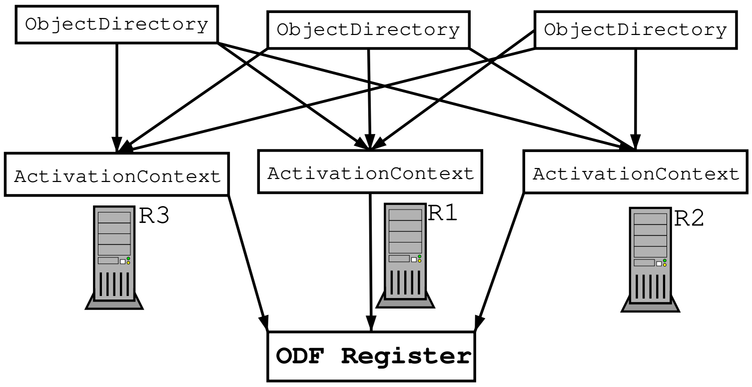
5.2 Arhitektura sistema za porazdeljevanje bremena

Pri načrtovanju sistema ODF bomo uporabili v razdelku 4.5 opisani model za porazdeljevanje bremena v CORBA aplikacijah: na objektnem nivoju bo odjemalec podal ODF omejitve, ki jim mora objekt zadoščati. Te bodo podane v enaki obliki, kot jo pričakuje OAF, torej z uporabo povpraševalnega jezika OAF, opisanega zgoraj. ODF bo odjemalcu vrnil referenco enega od objektov, ki ustreza omejitvam. Dejansko bo porazdeljevanje potekalo na sistemskem nivoju: ODF bo odjemalcu vrnil referenco objekta, ki ga realizira strežnik (proces) na najustreznejšem (po kriterijih uporabljene porazdeljevalne strategije) računalniku.

Sistem ODF bo sestavljen iz dveh vmesnikov (oziroma iz dveh primerkov CORBA objektov, ki bosta izvedla ta vmesnika): prvi, vmesnik `Register`, skrbi za seznam

[4]OD in AC ločimo ne le na osnovi imena računalnika, kjer tečejo, temveč še na osnovi imena uporabnika, ki ju uporablja, imena seje (angl. session) ipd., vendar bomo to zavoljo preprostosti prezrli.

AC in OD na vseh računalnikih omrežja, ki jih bomo uporabljali pri porazdeljevanju bremena. Njegov osnovni namen bo služiti kot registracijsko mesto za OAF in mu posredovati reference AC in OD. Poleg tega bo skrbel tudi za aktiviranje objektov AC in OD na oddaljenih računalnikih. Vlogo objekta `Register` prikazuje slika 5.3.



Slika 5.3: Objekt Register

Drugi vmesnik, `Distributor`, ki bo opravljal dejansko porazdeljevanje, bo vseboval eno samo metodo, in sicer `select_server`. Tej bo odjemalec podal seznam vseh strežnikov, ki realizirajo objekt z želenimi lastnostmi, vrnila pa mu bo najprimernejši strežnik iz seznama.

Opis obeh vmesnikov v jeziku IDL je naslednji:

```
module ODF
{
        typedef sequence<string> HostList;

        interface Register {
                exception HostFileError {
                        string info;
                };
                exception AccessError {
                        string info;
                };
                exception DuplicateHost {};
                exception UnknownHost {};
                exception NotRegistered {};
                exception BootstrapFailed {
                        HostList failed_hosts;
                };

                Object get_service(in string service_name, in string host_name,
                                   in string domain);

                void register_service(in string service_name, in string host_name,
                                      in string domain, in Object service)
```

```
                        raises (AccessError);
                void unregister_service(in Object service)
                        raises (AccessError, NotRegistered);

                void lock() raises (AccessError);
                oneway void unlock() raises (AccessError);

                readonly attribute HostList host_list;

                void add_hosts_from_file(in string filename)
                        raises (AccessError, BootstrapFailed);
                void add_host(in string hostname)
                        raises (AccessError, DuplicateHost, BootstrapFailed);
                void remove_host(in string hostname)
                        raises (AccessError, UnknownHost);
        };

        interface Distributor
        {
                exception NoServer {
                        string reason;
                };

                OAF::ServerInfo select_server(in Register reg,
                                              in OAF::ServerInfoList server_list)
                        raises (NoServer);
        };
};
```

5.2.1 Vmesnik Register

Kot je bilo že omenjeno je osnovni namen vmesnika `Register` služiti kot registracijsko mesto OAF. Preslikavo med metodami vmesnika `Register` in funkcijami, ki jih mora opravljati registracijsko mesto (glej razdelek 5.1.2), podaja tabela 5.1.

Funkcija registracijskega mesta	Metoda vmesnika `Register`
`check`	`get_service`
`register`	`register_service`
`unregister`	`unregister_service`
`lock`	`lock`
`unlock`	`unlock`

Tabela 5.1: Preslikava med metodami vmesnika `Register` in funkcijami registracijskega mesta

V primeru, ko dostop do registracijskega mesta ni dovoljen (registracijsko mesto je zaklenjeno), bodo ustrezne metode sprožile izjemo `AccessError`.

Za uporabo vsakega računalnika v omrežju moramo na njem aktivirati AC in OD ter registrirati njuni referenci. Omogočiti moramo tudi dostop do izvedb objektov na tem računalniku odjemalcem na vseh ostalih računalnikih, kar dosežemo tako, da dodamo OD vsem registriranim ACjem. Vmesnik `Register` vsebuje še tri metode, ki olajšajo to početje:

`add_hosts_from_file` prebere seznam imen računalnikov iz datoteke in za vsakega pokliče metodo `add_host`

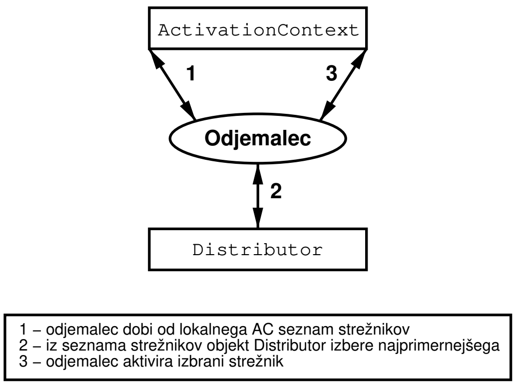
`add_host` požene `oafd` na oddaljenem računalniku, registrira AC in OD, ki ju ta izvede, ter doda novi OD vsem registriranim ACjem

`remove_host` odstrani referenci AC in OD na tem računalniku iz tekočega seznama ter odvzame OD vsem še registriranim ACjem

V primeru, ko poganjanje strežnika `oafd` na oddaljenem računalniku ne uspe, sprožita metodi `add_host` in `add_hosts_from_file` izjemo `BootstrapFailed`, saj tedaj `Register` ne more do referenc OD in AC.

5.2.2 Vmesnik Distributor

Vmesnik `Distributor` je namenjen izvedbi poljubne porazdeljevalne strategije. Odjemalec z uporabo metode `query` lokalnega AC (glede na odjemalca) dobi seznam strežnikov, ki ustrezajo njegovim omejitvam Tega poda metodi `select_server` neke izvedbe vmesnika `Distributor` [5] kot parameter, ta pa mu na osnovi porazdeljevalne strategije, ki jo izvaja, vrne najprimernejši strežnik. Odjemalec nato z OAF aktivira objekt, ki ga realizira vrnjeni strežnik. Pri tem predpostavljamo, da ID izvedbe strežnika (`iid` v opisni datoteki) in ime računalnika, na katerem se nahaja strežnik, le-tega enolično določata. Ta postopek aktivacije s sistemom ODF prikazuje slika 5.4.



Slika 5.4: Aktivacija objekta z uporabo sistema ODF

[5] Izvedbe vmesnika Distributor bomo v bodoče zavoljo jasnosti imenovali kar *porazdeljevalniki.*

Z uporabo ločenega objekta za izvedbo porazdeljevanja dosežemo, da lahko uporabnik sam nadgrajuje sistem ODF z lastnimi porazdeljevalnimi strategijami, če le-te zavije v izvedbo vmesnika `Distributor`.

Porazdeljevalniki so običajni CORBA objekti, katerih reference dobi odjemalec s pomočjo ODF. Vsak porazdeljevalnik realizira vmesnik `Distributor`, v datoteko, ki opisuje porazdeljevalnik, pa bomo za določanje izvedene porazdeljevalne strategije dodali še atribut `distribution_policy`. Na njegovi osnovi bo lahko odjemalec aktiviral porazdeljevalnik z želeno strategijo. Opis porazdeljevalnika s krožno (angl. round-robin) porazdeljevalno strategijo, kakršnega bomo opisali pozneje, je videti takole:

```
<oaf_server iid="OAFIID:ODF_Distributor_RR:Jaka" type="exe"
            location="odfd-roundrobin">
        <oaf_attribute name="repo_ids" type="stringv">
                <item value="IDL:ODF/Distributor:1.0"/>
        </oaf_attribute>
        <oaf_attribute name="distribution_policy" type="string"
                       value="round-robin"/>
</oaf_server>
```

5.2.3 Aktivacija s porazdeljevanjem

Odjemalec bo aktivacijo že znanega objekta z vmesnikom `Kalkulator` z uporabo krožne porazdeljevalne strategije izvedel takole:

```
// aktivirajmo porazdeljevalnik
requirements = "repo_ids.has('IDL:ODF/Distributor:1.0') AND " \
               "distribution_policy == 'round-robin'"
distributor = AC::activate(requirements, ...);

// AC povprašamo po seznamu strežnikov, ki realizirajo vmesnik kalkulator
requirements = "repo_ids.has('IDL:Primer/Kalkulator:1.0')";
server_list = AC::query(requirements, ...);

// naj porazdeljevalnik izbere primeren strežnik
server = distributor::select_server(register, server_list);

// aktivirajmo objekt v izbranem strežniku
requirements = "iid == '" + server.iid + "' AND " \
               "hostname == '" + server.hostname + "'";
kalkulator = AC::activate(requirements, ...);
```

Dodatno delo v primerjavi s preprosto aktivacijo, ki smo jo prikazali v razdelku 5.1.2, je v dveh dodatnih klicih metod `query` in `select_server`. Čas, ki ga zahteva klic metode `select_server`, je odvisen od uporabljene porazdeljevalne strategije, klic metode `query` pa je v istem velikostnem razredu kot klic metode `activate`. Čas za aktivacijo bo torej vsaj dvakrat daljši kot pri preprosti aktivaciji brez ODF, seveda pa nanj vpliva tudi lokacija porazdeljevalnika in omrežna komunikacija, ki je v zvezi z njo potrebna za klic njegovih metod. Objekt AC je vedno na (za odjemalca) lokalnem računalniku, zato na čas, potreben za klic njegovih metod, stanje omrežnih povezav ne vpliva. Porazdeljevalnik moramo aktivirati le enkrat, tako pridobljeno referenco pa uporabljamo za vsa nadaljnja porazdeljevanja.

5.3 Izvedba sistema ODF

Najprej bomo opisali izvedbo vmesnika `Register`, nato pa še izvedbo štirih porazdeljevalnikov z vmesnikom `Distributor`, ki bodo izvajali različne porazdeljevalne strategije. Vse izvedbe so namenjene različicam operacijskega sistema UNIX[6], napisane pa so bile v programskem jeziku C. Poleg samega sistema smo po vzoru OAF izvedli v jeziku C tudi knjižnico, poimenujmo jo `libodf`, za lažjo uporabo sistema, dinamično proženje operacij ter uporabo zakasnjenih sinhronih operacij, kar vse bomo potrebovali za vzporedno programiranje. K odločitvi za C je, nerodni preslikavi jezika IDL v C[7] navkljub, pripomoglo predvsem to, da je kodo, pisano v jeziku C, lahko uporabiti v večini ostalih programskih jezikov, npr. v Perlu. Tako bomo knjižnico `libodf` brez težav uporabljali v večini programskih jezikov.

5.3.1 Izvedba vmesnika Register

Vmesnik register bo realiziral strežnik `odf-regd`, ki bo centralno skrbel za porazdeljevanje bremena. Sistem ODF uporablja več datotek, ki jih najde v imeniku `.odf` v domačem imeniku uporabnika, ki ga poganja. Ob zagonu strežnik prebere konfiguracijsko datoteko `hosts` , ki vsebuje seznam imen vseh računalnikov. Datoteka je zapisana v jeziku XML, na primer:

```
<odfhostlist>
        <host name="machiavelli.ijs.si"/>
        <host name="artur.ijs.si"/>
        <host name="pluton.ijs.si"/>
        <host name="coke.ijs.si"/>
</odfhostlist>
```

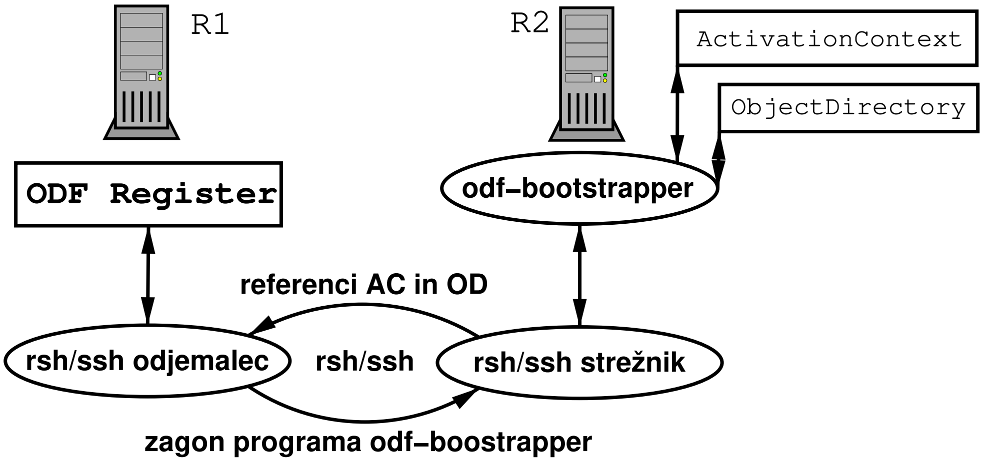
Na vsakem od računalnikov poizkuša strežnik aktivirati AC in OD. Aktivacija poteka z izvajanjem pomožnega programa `odf-bootstrapper`, ki mora biti nameščen na oddaljenem računalniku. Strežnik požene `odf-bootstrapper` s pomočjo protokola *rsh* ali *ssh*[8]. Na oddaljenem računalniku `odf-boostrapper` najprej izvede vse ukaze v datoteki `login`, aktivira AC in OD ter njuni referenci izpiše na

[6] Razvite so bile na operacijskem sistemu Linux, preizkušene pa tudi na SunOS 5.6 ter HP-UX 10 in 11.

[7] C sintaktično ne podpira objektno usmerjenega programiranja.

[8] *Rsh* ali *Remote Shell* je protokol, ki se uporablja za izvajanje ukazov v ukazni lupini na oddaljenem računalniku. *Ssh* ali *Secure Shell* ima enak namen, le da je komunikacija med računalnikoma kriptirana.

svoj standardni izhod, ki ga preko protokola *rsh* oziroma *ssh* bere strežnik. Ta proces prikazuje slika 5.5



Slika 5.5: Registracija oddaljenih AC in OD

Strežnik je pripravljen, ko je opravljena aktivacija AC in OD na vseh računalnikih, navedenih v datoteki `hosts`. Zgodi se, da aktivacija na nekaterih računalnikih ne uspe v določenem času; tedaj strežnik prekine proces aktivacije na teh računalnikih ter deluje le z naborom računalnikov, na katerih je aktivacija uspela. Po končanem zagonu in registraciji referenc AC in OD, strežnik izpiše svojo referenco v datoteko `ior` in čaka na odjemalce. Najpomembnejši odjemalec bo OAF, ki ga bo uporabljal kot registracijsko mesto, ostali odjemalci pa lahko tudi po zagonu poljubno dodajajo ali odstranjujejo računalnike z uporabo metod `add_host`, `add_host_from_file` in `remove_host`.

5.3.2 Izvedba porazdeljevalnikov

Za preizkušanje sistema ODF bomo razvili porazdeljevalnike, ki bodo izvedli štiri porazdeljevalne strategije. Vsaka strategija izbere za aktivacijo iz seznama strežnikov $S = \{S_0, S_1, ...S_{N_S-1}\}$ nek strežnik S_D. Vsak strežnik $S_D \in S$ je nameščen na natanko enem računalniku $R_D \in R = \{R_0, R_1, ...R_{N_R-1}\}$. Izvedli bomo naslednje porazdeljevalne strategije:

- *naključna strategija* izbere naključen strežnik iz S
- *krožna (angl. round-robin) strategija* izbere za aktivacijo z zaporedno številko M strežnik na računalniku $R_D = R_{M\ mod\ N_R}$, kjer je N_R je število vseh računalnikov
- *utežena strategija* nastavi vsakemu od računalnikov $R_D \in R$ utež W_D. Za aktivacijo M uporabi strežnik na računalniku R_D z verjetnostjo $P_D = W_D/W$, kjer je $W = \sum_{n=0}^{N_R-1} W_n$

- *pragovna strategija (angl. load thresholding)* meri trenutno obremenitev (load) $L_D \in [0, 1]$ računalnika R_D. Za aktivacijo uporabi prvi strežnik, ki se nahaja na računalniku, s trenutno obremenitvijo $L_D < L_T$, kjer je L_T prag obremenitve

Izvedbi naključne in krožne strategije sta očitni, komentirajmo le zadnji dve:

Utežena strategija

Za izvedbo utežene strategije definiramo nov vmesnik `Weighted`, ki deduje iz vmesnika `Distributor`:

```
module ODF
{
        interface Weighted : Distributor
        {
                void set_weight(in string hostname, in float weight);
        };
};
```

Metodo `set_weight` odjemalec uporabi, da nastavi utež računalnika z imenom `hostname` na vrednost `weight`. Določanje uteži je prepuščeno odjemalcu, privzete vrednosti v naši izvedbi porazdeljevalnika so 1.

Pragovna strategija

Tudi za izvedbo te strategije definiramo nov vmesnik `LoadThresh`, izveden iz vmesnika `Distributor`:

```
module ODF
{
        interface LoadThresh : Distributor
        {
                void set_threshold(in float threshold);
                void set_host_threshold(in string hostname, in float threshold);
        };
};
```

Metoda `set_threshold` nastavi prag obremenitve za vse računalnike, metoda `set_host_threshold` pa prag obremenitve za računalnik z imenom `hostname`.

Porazdeljevalnik se v metodi `select_server` sprehodi po vseh strežnikih v seznamu ter izmeri obremenitev računalnika, na katerem je strežnik nameščen, in jo primerja s pragom obremenitve za podani računalnik, če je ta nastavljen, sicer pa s pragom za vse računalnike. Strežnik S_D je primeren za aktivacijo, če je izmerjena obremenitev manjša od praga obremenitve. V primeru, ko obremenitev presega prag na vseh računalnikih, metoda `select_server` sproži izjemo `NoServer`. Nastavljanje pragov je prepuščeno odjemalcu.

Postavlja se še vprašanje, kako definirati obremenitev računalnikov. Za naše namene bomo za kazalec obremenitve uporabili že v razdelku 4.4 omenjeno izkoriščenost računalnika. Obremenitev bomo definirali kot realno število $L \in [0, 1]$, ki pove, kolikšen delež zadnje sekunde je bila CPE zaposlena.

Za lažje merjenje obremenitve na oddaljenih računalnikih bomo tudi merilec obremenitve izvedli kot CORBA objekt z vmesnikom `Overseer::CPU` :

```
module Overseer {
        interface CPU {
                float get_idle();
                float get_system();
                float get_user();
        };
};
```

Porazdeljevalnik bo aktiviral merilec obremenitve na želenem računalniku s pomočjo OAF in uporabil metodo `get_idle`, ki vrne delež časa T_M, ko je CPE mirovala (idle). Obremenitev lahko zdaj izračunamo: $L = 1 - T_M$.

Koda za merjenje obremenitve se razlikuje že med različicami operacijskega sistema UNIX. Za potrebe testiranja smo merjenje obremenitve izvedli za operacijske sisteme Linux, SunOS in HP-UX.

5.3.3 Upravljanje sistema ODF

Pregled statusa sistema ODF in ravnanje z njim bomo izvedli z uporabo programa `odf-control` , ki je namenjen uporabi iz ukazne vrstice. Z naslednjimi parametri lahko pregledujemo in spreminjamo trenutno stanje sistema:

`--hosts` izpiše seznam vseh registriranih računalnikov

`--load` izpiše poleg imen registriranih računalnikov še njihovo trenutno obremenitev, ki jo izmeri s pomočjo nameščenih objektov `Overseer::CPU`

`--register=<ime>` registrira nov računalnik z imenom `<ime>`

`--unregister=<ime>` odstrani iz seznama računalnik imenovan `<ime>`

`--dirs` poleg imen registriranih računalnikov izpiše še imena računalnikov, na katerih so objekti OD, ki jih ta računalnik pozna

Pregledovanje in celo spreminjanje stanja sistema je mogoče tudi tedaj, ko ga že uporablja poljubno število aplikacij.

5.4 Knjižnica libodf

Namen knjižnice `libodf` je olajšati uporabo sistema ODF in vzporedno programiranje s CORBO v jeziku C. Knjižnica skrbi tudi za prepoznavanje standardnih parametrov ukazne vrstice, ki naj bi jih podpirale vse aplikacije, ki uporabljajo sistem ODF. S shranjevanjem nekaterih pogosto uporabljanih vrednosti na strani same aplikacije poskrbi tudi za drastično zmanjšanje komunikacije med aplikacijo ter AC in množico OD, ki je potrebna za porazdeljevanje.

Knjižnica vsebuje dve skupini funkcij:

- funkcije, ki skrbijo za uporabo sistema ODF
- funkcije za uporabo dinamičnega izstavljanja zahtev: dinamično ustvarjanje zahtev, njihovo pošiljanje ter čakanje na odgovor

Oglejmo si najprej programski vmesnik, ki ga knjižnica razkriva programerju, nato pa še podroben opis posameznih tipov in funkcij, ki ga ta definira:

```
typedef enum
{
        ODF_FLAG_IGNORE_LOCAL    = (1 << 0),
        ODF_FLAG_IGNORE_REMOTE   = (1 << 1),
        ODF_FLAG_WAIT_FOR_SERVER = (1 << 2)
} ODFSelectionFlags;

typedef struct _ODFRequestInfo ODFRequestInfo;

struct _ODFRequestInfo
{
        CORBA_Request req;
        CORBA_Object server;
        OAF_ActivationID aid;
        CORBA_NamedValue *result;
        CORBA_NVList *args;
        gboolean no_response;
        guint id;
};

extern struct poptOption odf_popt_options[];

/* initialize ODF */
CORBA_ORB odf_init(int *argc, char **argv);

/* obtain a reference to the IDL:ODF/Register:1.0 object used by ODF */
ODF_Register odf_get_register(void);

/* obtain a reference to a Distributor object by its distribution policy */
ODF_Distributor odf_get_distributor(gchar *policy, CORBA_Environment *ev);

/* object distribution/activation */
```

```
CORBA_Object odf_activate(ODF_Distributor distributor, const char *requirements,
                          char *const *selection_order, ODFSelectionFlags selflags,
                          OAF_ActivationFlags flags, OAF_ActivationID *ret_aid,
                          CORBA_Environment *ev);

/* ODFRequestInfo convenience functions */
ODFRequestInfo *odf_request_info_new(CORBA_Request req, CORBA_Object server,
                                     CORBA_NamedValue *res, CORBA_NVList *args,
                                     OAF_ActivationID aid, CORBA_Environment *ev);
ODFRequestInfo *odf_request_info_create(CORBA_ORB orb, CORBA_Environment *ev,
                                        CORBA_Context ctx,  OAF_ActivationID aid,
                                        CORBA_Object obj, gchar *op_name,
                                        CORBA_Flags req_flags,
                                        CORBA_TypeCode res_tc,
                                        CORBA_unsigned_long res_len,
                                        gpointer res_val,
                                        guint par_count, ...);
void odf_request_info_free(ODFRequestInfo *ri, CORBA_Environment *ev);

/* send the request associated with the ODFRequestInfo */
void odf_request_send(ODFRequestInfo *ri, gboolean expect_response,
                      CORBA_Environment *ev);
/* send immediately */
void odf_request_send_now(ODFRequestInfo *ri, CORBA_Environment *ev);

/* poll for reply to a single request */
gboolean odf_request_poll(ODFRequestInfo *ri, gboolean block,
                          CORBA_Environment *ev);
/* poll for replies to a list of requests */
ODFRequestInfo *odf_request_poll_list(GList *req_list, gboolean block,
                                      CORBA_Environment *ev);

/* retrieve the list of all requests that haven't been replied to yet */
GList *odf_get_pending_requests();
```

5.4.1 Podatkovni tipi

Oštevilčenje (angl. enumeration) `ODFSelectionFlags` je namenjeno dodatnemu nadzoru nad aktivacijo objektov s funkcijo `odf_activate`:

`ODF_FLAG_IGNORE_LOCAL` povzroči, da porazdeljevalnik ne upošteva strežnikov na lokalnem računalniku

`ODF_FLAG_IGNORE_REMOTE` povzroči, da porazdeljevalnik upošteva le strežnike na lokalnem računalniku

`ODF_FLAG_WAIT_FOR_SERVER` zahteva, da v primeru, ko porazdeljevalnik ne najde ustreznega strežnika, funkcija ne vrne napake, temveč poizkuša z aktivacijo, dokler ta ne uspe

Struktura `ODFRequestInfo` hrani vse podatke, ki so pomembni za posamezno izstavljeno zahtevo. Aplikaciji omogoča, da laže vodi seznam vseh zahtev, ki jih je razposlala, in zbira odgovore nanje, ko ti prispejo. Njeni člani enolično določajo zahtevo (`req`), njene parametre ter prostor za rezultat (`args` oziroma `result`), strežnik (`server` in `aid`) ter to, ali zahteva pričakuje odgovor (`no_response`).

5.4.2 Funkcije za uporabo ODF

Ta skupina funkcij poskrbi za inicializacijo ODF, aktivacijo porazdeljevalnikov in aktivacijo poljubnih vmesnikov z uporabo porazdeljevalnika.

`odf_init` inicializira sistem ODF: najprej poskrbimo za morebitne parametre iz ukazne vrstice. Vse aplikacije, ki uporabljajo `libodf`, prepoznajo dva standardna parametra:

- `--odf-register-ior` dovoli, da v ukazni vrstici navedemo referenco objekta `Register`, ki ga bo uporabljala aplikacija
- `--odf-cache` omogoči shranjevanje seznamov strežnikov in referenc strežnikov na strani aplikacije, s čimer zmanjšamo potrebno komunikacijo med objekti. To lastnost bomo podrobneje razložili kasneje

Nato knjižnica inicializira ORB in OAF ter pridobi referenco objekta `Register` na naslednji način: če je referenca podana v ukazni vrstici, uporabi to, sicer pa poskuša prebrati referenco iz datoteke `ior`, kamor jo zapiše `Register` ob zagonu. Če tudi to ne uspe, poizkusi aktivirati objekt Register z uporabo OAF. V tem primeru se, če je to potrebno, požene tudi nov strežnik, ki realizira vmesnik `Register`. Na koncu objekt `Register` dodamo OAF kot registracijsko mesto

`odf_get_register` vrne referenco registra

`odf_get_distributor` aktivira porazdeljevalnik, ki uporablja porazdeljevalno strategijo `policy`

`odf_activate` skrbi za aktivacijo s porazdeljevanjem. Vrne referenco aktiviranega objekta, ki ustreza omejitvam `requirements`, strežnik pa je izbran s porazdeljevalnikom `distributor`. Parameter `selflags` ima lahko postavljen poljuben nabor zastavic iz oštevilčenja `ODFSelectionFlags`, s katerimi dodatno nadzorujemo, kako bo izbran strežnik. Poleg reference funkcija v parametru `ret_aid` vrne tudi *identifikacijski niz* (Activation ID) , ki enolično določa aktivirani strežnik. Tega bomo hranili v strukturi `ODFRequestInfo` in ga uporabljali za povezavo med izstavljenimi zahtevami in strežniki. Pomen ostalih parametrov je enak kot pri aktivaciji z OAF (glej razdelek 5.1.2)

5.4.3 Funkcije za vzporedno programiranje

Ostanek funkcij iz naše knjižnice je namenjen predvsem lažjemu dinamičnemu ustvarjanju zahtev, njihovemu izstavljanju in zbiranju odgovorov:

`odf_request_info_new` zaseže pomnilnik za novo strukturo `ODFRequestInfo` in inicializira njene člane s posredovanimi vrednostmi

`odf_request_info_create` ima podobno funkcijo, vendar sama zaseže prostor za parametre in rezultat ter ustvari zahtevo. Namesto nerodnega ustvarjanja zahteve, kot smo ga opisali v razdelku 3.7.2, lahko zahtevo in njej pripadajoči primerek strukture `ODFRequestInfo` dinamično ustvarimo s klicem ene same funkcije:

```
CORBA_long tri, a = 1, b = 2;
ri = odf_request_info_create(
        orb, ev, ctx,    /* reference ORBa, izjeme in konteksta */
        aid,             /* identifikacijski niz strežnika */
        server,          /* referenca le-tega */
        "sestej",        /* ime operacije */
        0,               /* zastavice zahteve */
        TC_long,         /* koda tipa rezultata */
        sizeof(tri),     /* dolzina rezultata */
        &tri,            /* kazalec na rezultat */
        2,               /* stevilo parametrov */
        "s1",            /* ime parametra */
        TC_long,         /* koda tipa parametra */
        sizeof(a),       /* dolzina parametra */
        &a,              /* kazalec na parameter */
        0,               /* zastavice parametra */
        "s2",            /* ime parametra */
        TC_long,         /* koda tipa parametra */
        sizeof(b),       /* dolzina parametra */
        &b,              /* kazalec na parameter */
        0                /* zastavice parametra */
        );
```

`odf_request_info_free` sprosti pomnilnik, zasežen za strukturo `ODFRequestInfo` ter njej pripadajočo zahtevo in referenco strežnika

`odf_request_send` pošlje zahtevo, ki jo opisuje parameter `ri`; če aplikacija pričakuje rezultat naj ima parameter `expect_response` vrednost `TRUE`, sicer pa `FALSE`. V primeru, ko strežnik, ki naj mu bo zahteva poslana, že obdeluje neko drugo zahtevo naše aplikacije, bo knjižnica zahtevo postavila v seznam in jo poslala šele, ko bo strežnik na voljo

`odf_request_send_now` pošlje zahtevo takoj, četudi je strežnik morda zaseden `odf_request_poll` preveri, če je ORB že dobil odgovor na zahtevo, ki jo opisuje parameter `ri`. Parameter `block` opisuje obnašanje funkcije, kadar odgovora ni na voljo: vrednost `TRUE` povzroči, da funkcija zaustavi izvajanje programa, dokler odgovor ne prispe, sicer funckija vrne vrednost `FALSE`, izvajanje glavnega programa pa se nadaljuje

`odf_request_poll_list` ima podoben namen, vendar preverja, če je ORB dobil odgovor na vsaj eno zahtevo iz seznama `req_list`. V tem primeru vrne opis prve zahteve iz seznama, na katero je prispel odgovor

`odf_get_pending_requests` vrne seznam vseh poslanih zahtev, na katere aplikacija še ni prejela odgovora

5.4.4 Predpomnjenje seznamov strežnikov in referenc

Kot bomo videli v razdelku 6.4.2, je pri velikem številu računalnikov in na njih nameščenih objektov aktivacija lahko zelo zamudna: zanjo je najprej potrebna analiza opisa omejitev z LALR(1) sintaktičnim analizatorjem, nato pridobivanje seznamov strežnikov od vseh OD in končno še sestavljanje seznama strežnikov, ki ustrezajo navedenim omejitvam.

Čas, porabljen za aktivacijo, lahko mirno zanemarimo, kadar je ta majhen v primerjavi s časom, porabljenim za streženje zahtevi. Tega ne moremo zagotoviti vedno, saj pogosto ni mogoče poljubno določati velikosti problemov, ki jih bo rešila posamezna zahteva. Tudi kadar je to mogoče, pa moramo, če naj izpolnimo zahtevo, da naj sistem deluje na poljubnem omrežju (glej razdelek 1), velikost problemov določati med izvajanjem aplikacije in ne morebiti ob prevajanju, saj je čas, potreben za aktivacijo, odvisen od števila računalnikov v omrežju in na njih nameščenih objektov ter zato znan šele ob izvajanju aplikacije na konkretnem omrežju.

Premislek pokaže, da aplikacije običajno izvedejo veliko aktivacij, vendar aktivirajo le majhno (v primerjavi s številom vseh aktivacij) število različnih strežnikov. Na nivoju same aplikacije se to kaže v majhni moči množice omejitev, katerim naj aktivirani strežniki zadoščajo.

Vsakemu od opisov omejitev ustreza enak seznam strežnikov. Tega knjižnica prvič pridobi od AC z metodo `query`, shrani na strani aplikacije in ga naslednjič, ko aplikacija zahteva aktivacijo strežnika z enakimi lastnostmi, uporabi, ne da bi spet klicala metodo `query`. To močno zmanjša čas za vse naslednje aktivacije.

Aktivacija strežnika, ki ga iz seznama vseh ustreznih izbere porazdeljevalnik, spet zahteva komunikacijo z AC: poklicati moramo metodo `activate`, ki vrne referenco strežnika. Tudi te reference knjižnica hrani na aplikacijski strani ter jih ponovno uporabi, ko porazdeljevalnik izbere isti strežnik. Tako prihranimo še čas, potreben za klic metode `activate`.

Predpomnjenje seznamov in referenc na strani aplikacije omogočimo s parametrom `--odf-cache`, ki ga navedemo v ukazni vrstici, ko zaganjamo aplikacijo. Pospešitev, ki jo prinese uporaba te rešitve je prikazana v razdelku 6.3.

5.5 Primer aplikacije

Za konec si oglejmo še primer preproste aplikacije, ki uporablja ODF preko knjižnice `libodf`. Aplikacija bo s pomočjo množice objektov z že dobro znanim vmesnikom `Kalkulator` vzporedno seštela seznam parov števil:

```
#include <orb/orb.h>
#include <orb/corba_typecode.h>
#include <liboaf/liboaf.h>
#include <libodf/libodf.h>

int
main(int argc, char **argv)
{
        /* dva seznama: prvi hrani pare stevil, drugi bo hranil njihove vsote */
        CORBA_long array[NUM_PAIRS][2] = { ... }, res[NUM_PAIRS];

        CORBA_Environment ev;
        CORBA_ORB orb;
        CORBA_Context ctx;
        CORBA_Object distributor, kalkulator;
        ODFRequestInfo *ri;
        GList *rlist = NULL;
        OAF_ActivationID aid;

        /* inicializiramo okolje za prenasanje izjem. zavoljo jasnosti
           bomo izjeme v nadaljevanju prezrli. */
        CORBA_exception_init(&ev);

        /* inicializiramo ODF; funkcija vrne referenco uporabljanega ORBa */
        orb = odf_init(&argc, argv);

        /* potrebujemo se referenco CORBA Context objekta... */
        CORBA_ORB_get_default_context(orb, &ctx, &ev);

        /* ...in referenco porazdeljevalnika */
        distributor = odf_get_distributor("round-robin", &ev);

        /* za vsak par stevil... */
        for(i = 0; i < NUM_PAIRS; i++) {
                /* ...aktiviramo objekt z vmesnikom Kalkulator... */
                kalkulator = odf_activate(distributor,
                        "repo_ids.has('IDL:Primer/Kalkulator:1.0')",
                        NULL, 0, 0, &aid, &ev
                        );
                /* ...ustvarimo novo zahtevo... */
                ri = odf_request_info_create(orb, &ev, ctx, aid, kalkulator,
                        "sestej", 0,
                        TC_long, sizeof(CORBA_long), &res[i],
                        2,
                        "s1", TC_long, sizeof(CORBA_LONG), &array[i][0],
```

```
                        "s2", TC_long, sizeof(CORBA_LONG), &array[i][1]
                        );
                /* ...jo dodamo seznamu poslanih zahtev... */
                rlist = g_list_prepend(rlist, ri);
                /* ...in jo posljemo */
                odf_request_send(ri, TRUE, &ev);
                /* sprostimo referenco kalkulatorja */
                CORBA_Object_release(kalkulator, &ev);
        }

        /* cakamo na rezultate */
        while(rlist != NULL) {
                /* blokiramo izvajanje dokler ne dobimo vsaj en rezultat */
                ri = odf_request_poll_list(rlist, TRUE, &ev);
                /* dobili smo rezultat: zahtevo odstranimo iz seznama */
                rlist = g_list_remove(rlist, ri);
        }

        /* tako, stevila so sesteta */

        /* se pocistimo za sabo... */
        CORBA_Object_release(distributor);
        CORBA_Object_release(orb);
        CORBA_exception_free(&ev);

        return 0;
}
```

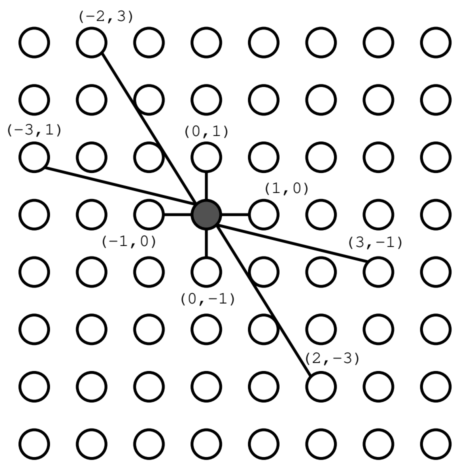
6 Preizkus sistema ODF

V tem poglavju predstavimo preizkusno aplikacijo ter dve omrežji, na katerih testiramo sistem ODF. Sistem preizkusimo na heterogenem omrežju z vozlišči, ki se razlikujejo tako po procesorski arhitekturi kot po nameščenem operacijskem sistemu, in na homogenem omrežju z enakimi vozlišči, kar nam omogoča analizo pospešitev v odvisnosti od števila vozlišč. Preizkusimo tudi učinkovitost predpomnjenja referenc.

6.1 Preizkusna aplikacija

Preizkusna aplikacija išče optimalno[1] postavitev povezav med vozlišči v dani mreži velikosti $X \times Y$ vozlišč stopnje D, kar je pogost problem pri načrtovanju masovno vzporednih računalnikov.

Eno od optimalnih postavitev v mreži velikosti 8×8 stopnje 8 prikazuje slika[2] 6.1.



Slika 6.1: Optimalna postavitev povezav

Problem bomo reševali z izčrpnim preiskovanjem prostora vseh možnih postavitev,

[1] Za optimalno postavitev imamo tisto, ki ima najmanjšo vsoto povprečne in maksimalne razdalje med poljubnima dvema vozliščema v mreži, gledano preko vseh možnih parov vozlišč.

[2] Zavoljo preglednosti so prikazane le povezave enega od procesorjev.

kar je časovno zelo zamudno: na enem samem računalniku s CPE AMD Athlon K7 s frekvenco 750 MHz lahko izvedeni algoritem v eni sekundi preišče približno 12 postavitev povezav v mreži velikosti 32×32 stopnje 10, kakršno smo uporabljali pri preizkusih. Pri takšni hitrosti bi vse možne postavitve v takšni mreži preiskali v dobrih 28 minutah. Pri večjih razsežnostih problema bi porabili še neprimerno daljši čas: v mreži velikosti 32×32 stopnje 12 bi iskanje trajalo 44 ur in 15 minut.

Prvotna aplikacija izvaja postopek preiskovanja tako, da eno za drugo generira možne postavitve povezav ter računa največjo razdaljo med poljubnima vozliščema in povprečno razdaljo med vsemi možnimi pari vozlišč. Postopek lahko enostavno paraleliziramo tako, da prostor vseh možnih postavitev povezav razdelimo na več podprostorov, ki jih preiskujemo vzporedno, saj je iskanje optimalnih povezav v nekem podprostoru neodvisno od iskanja v drugem podprostoru.

Za uporabo sistema ODF smo postopek iskanja optimalne postavitve zavili v CORBA objekt z vmesnikom `Test::MeshAnalyzer`:

```
module Test
{
        typedef sequence<unsigned long> Table;
        typedef Table Mesh;
        typedef sequence<Mesh> MeshList;

        interface MeshAnalyzer
        {
                attribute Mesh link_config;
                void init_link_config();
                long find_link_config(in long n);
                void set_parameters(in long sx, in long sy, in long deg);
                MeshList find_optimal(inout unsigned long count,
                                      inout float avg_dist,
                                      inout float max_dist);
        };
};
```

`Mesh` je podatkovni tip, ki predstavlja postavitev povezav. V mreži stopnje D je to D parov števil, ki predstavljajo relativne koordinate procesorjev, s katerimi je povezan vsak procesor v mreži. Postavitev iz slike 6.1 predstavimo z zaporedjem:

$$< (-2,3),(2,-3),(-3,1),(3,-1),(0,1),(0,-1),(1,0),(-1,0) >$$

`set_parameters` nastavi parametre mreže: velikost ter stopnjo povezanosti

`init_link_config` določi začetno postavitev povezav

`find_link_config` generira `n`-to naslednjo postavitev povezav

`find_optimal` poišče optimalno postavitev povezav v množici naslednjih `count` postavitev. Vrne seznam vseh optimalnih postavitev, povprečno razdaljo med vsemi možnimi pari vozlišč (`avg_dist`) in največjo razdaljo med poljubnima vozliščema, gledano preko vseh možnih parov vozlišč (`max_dist`)

Strežnik, ki izvede ta vmesnik, poimenujmo `odf-optmesh`.

Aplikacijo paraleliziramo tako, da jo izvedemo v obliki odjemalca, ki razdeli prostor vseh možnih postavitev v dani mreži na podprostore fiksne velikosti ter s pomočjo ODF na omrežnih računalnikih aktivira objekte z vmesnikom `MeshAnalyzer` (izvede jih strežnik `odf-optmesh`, nameščen na vsakem od teh računalnikov), ki iščejo optimalne postavitve v posameznih podprostorih.

6.2 Preizkusni omrežji

Aplikacijo smo preizkusili na dveh omrežjih. Homogeno, kjer imajo vsi računalniki enako strojno in programsko opremo, nam je omogočilo predvsem analizo učinkovitosti sistema ODF v odvisnosti od števila računalnikov v omrežju. V heterogenem omrežju z različnimi računalniki, ki imajo nameščene različne operacijske sisteme, pa se pokaže, da programerju ni potrebno skrbeti za razlike med strojno in programsko opremo računalnikov: za to poskrbita CORBA in sistem ODF.

Heterogeno omrežje tvori pet računalnikov, katerih strojno in programsko opremo prikazuje tabela 6.1.

Računalnik	CPE	RAM	OS
machiavelli	Intel PIII 450 MHz	128 MB	Linux 2.2.16
coke	Intel Celeron 466 MHz	64 MB	Linux 2.2.14
artur	HP PA-RISC 180 MHz	128 MB	HP-UX 10
pluton	Sun 4U Sparc Ultra-30	128 MB	SunOS 5.6
mreza	AMD Athlon 750 MHz	128 MB	Linux 2.2.17

Tabela 6.1: Konfiguracija heterogenega omrežja

Ti računalniki so del večjega lokalnega omrežja, na katerem je poleg prometa, ki ga ustvarja sistem ODF, še dodaten promet.

Homogeno omrežje tvori devet računalnikov s CPE AMD Athlon, delujočih na frekvenci 750 MHz, s 128 MB fizičnega pomnilnika in nameščenim operacijskim sistemom Linux[3].

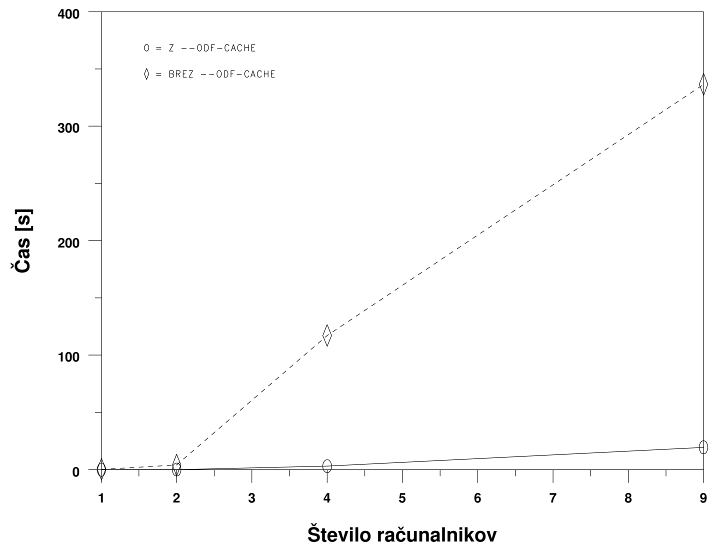
Računalniki homogenega omrežja so na privatnem omrežju, tako da je bil na homogenem omrežju edini promet tisti, ki ga je ustvarjala naša aplikacija.

6.3 Preizkus predpomnjenja referenc

V razdelku 5.4.4 smo razložili mehanizem predpomnjenja referenc, ki naj bi pripomogel k zmanjšanju časa, potrebnega za aktivacijo objektov; kot je razvidno iz slike

[3] Različica jedra 2.2.17.

6.2, se z večanjem števila računalnikov in s tem tudi množice nameščenih objektov, katerih seznam vodi OAF, čas, potreben za aktivacijo, hitro veča.



Slika 6.2: Čas za aktivacijo

Čas, ki ga potrebujemo za aktivacijo pri uporabi predpomnjenja referenc, je mnogo manjši, saj se celoten postopek aktivacije izvede le ob prvi aktivaciji, pri vseh naslednjih pa se uporabi shranjeni seznam strežnikov oziroma referenca že aktiviranega objekta.

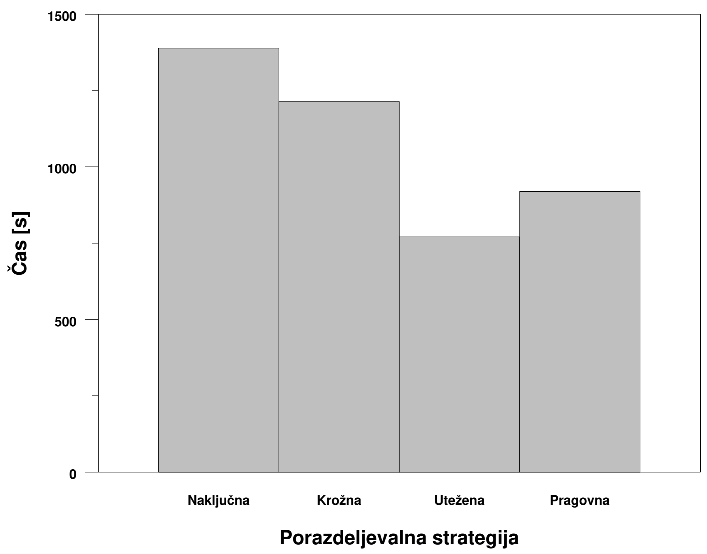
Pri uporabi predpomnjenja so aplikacije v mnogo manjši meri odvisne od velikosti posameznih zahtev, ki jih izstavljajo aktiviranim objektom. Kadar predpomnjenja ne uporabljamo, se na omrežjih z velikim številom računalnikov prav lahko zgodi, da je čas, potreben za aktivacijo, v istem velikostnem razredu kot čas, ki ga objekti potrebujejo za procesiranje zahtev. V tem primeru strežnik zahtevo izpolni, še preden so izstavljene zahteve strežnikom na ostalih računalnikih: aplikacija nikoli ne izkorišča vseh računalnikov naenkrat, temveč uporablja le neko njihovo podmnožico. Primer takšnega obnašanja bomo predstavili v razdelku 6.4.2.

6.4 Preizkus porazdeljevalnih strategij

V tem razdelku bomo predstavili preizkus štirih izvedenih porazdeljevalnih strategije, ki smo jih opisali v razdelku 5.3.2 na obeh preizkusnih omrežjih. Iskali bomo optimalno postavitev povezav v mreži velikosti 32×32 stopnje 10.

6.4.1 Heterogeno omrežje

Primerjava časa, potrebnega za izvrševanje aplikacije pri uporabi različnih porazdeljevalnih strategij in predstavljenega na sliki 6.3, pokaže, da je bila najučinkovitejša utežena porazdeljevalna strategija. Uteži, ki so bile nastavljene posameznim računalnikom, prikazuje tabela 6.2. Uteži so bile določene na osnovi časa, porabljenega za iskanje optimalne postavitve povezav v množici 10000 postavitev, in so zatorej prikrojene naši aplikaciji.



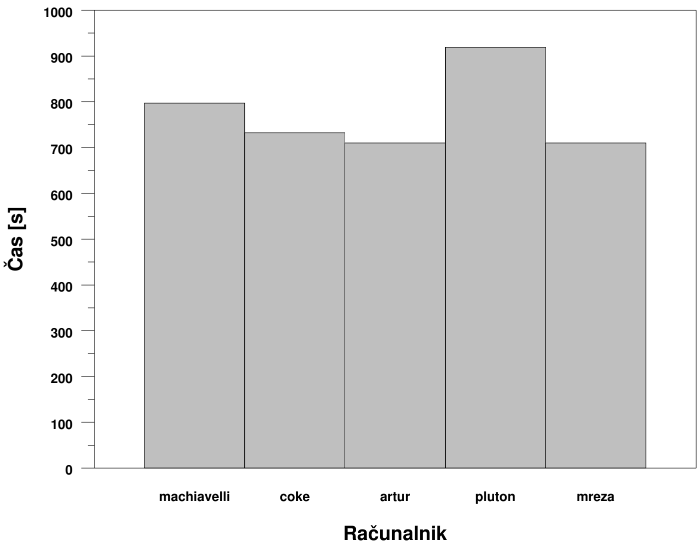
Slika 6.3: Čas računanja na heterogenem omrežju

Računalnik	Vrednost uteži
machiavelli	0,212
coke	0,194
artur	0,113
pluton	0,109
mreza	0,372

Tabela 6.2: Vrednosti uteži

Ostale strategije so se obnesle slabše. Težava krožne porazdeljevalne strategije je v tem, da se računalniki močno razlikujejo med sabo, vseeno pa jim strategija naloži enako število zahtev: hitrejši računalniki jih izvršijo prej, aplikacija pa mora čakati na počasnejše.

Edina dinamična porazdeljevalna strategija, pragovna strategija, se je sicer obnesla neprimerno bolje kot krožna, vendar pa dosti slabše kot utežena. Pričakovanje, da se bo spričo tega, da edina upošteva dejansko obremenitev računalnikov v trenutku aktivacije objekta, obnesla najbolje, se ne izpolni. Zakaj? Prvi razlog je v dodatnem delu in komunikaciji, ki sta potrebna za merjenje obremenitve: v najslabšem primeru je treba izmeriti obremenitev na vseh omrežnih računalnikih, v primeru, ko so vsi računalniki preobremenjeni, pa je potrebno z aktivacijo poizkusiti celo večkrat. Drugi, pomembnejši razlog se razkrije, če si ogledamo, koliko časa je zaposlen posamezen računalnik, kar nam pokaže slika 6.4. Sunova delovna postaja, `pluton`, porabi neprimerno več časa, kot ostali računalniki, analiza poteka izvajanja pa pokaže, da je pogosto dvakrat zapored aktiviran objekt na njej: to je posledica dejstva, da Sunovo jedro operacijskega sistema ažurira podatke o obremenitvi sistema redkeje kot ostala jedra. Porazdeljevalnik poizkusi v tem času večkrat aktivirati objekt in za aktivacijo vztrajno izbira `plutona`, dokler se ta informacija ne ažurira.



Slika 6.4: Čas računanja po računalnikih

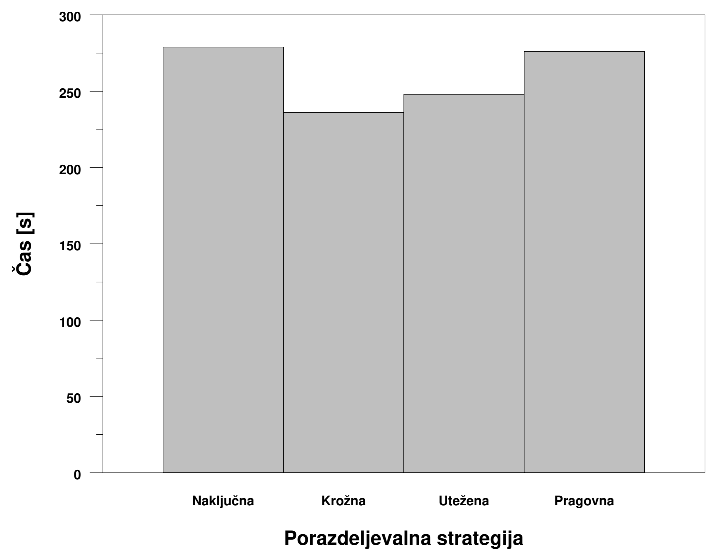
6.4.2 Homogeno omrežje

Na homogenem omrežju se je, kot vidimo na sliki 6.5, najbolje obnesla krožna porazdeljevalna strategija.

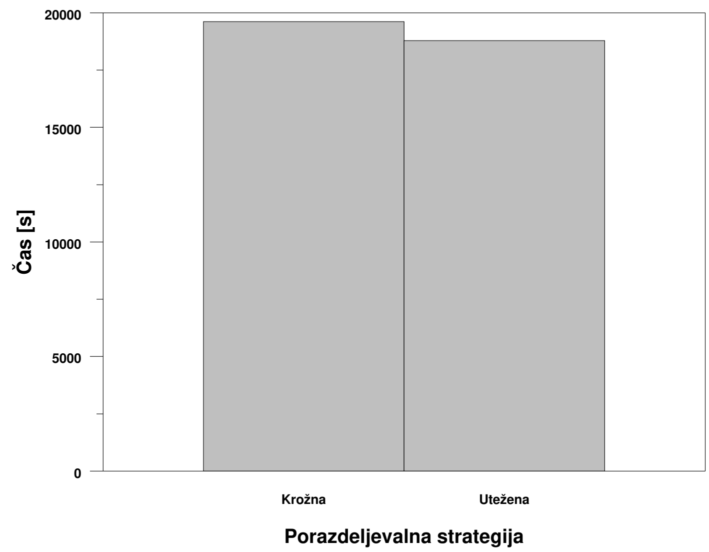
Utežena strategija se obnese malo slabše, razlog pa tiči v dejstvu, da je za preiskavo celotnega prostora postavitev potrebno izvesti le 204 aktivacije objektov, kar je relativno malo. Utežena strategija je po svoji naravi verjetnostna in zavoljo majhnega števila aktivacij ne doseže idealne porazdelitve objektov po računalnikih.

Pričakujemo, da se bo v primeru, ko je potrebnih več aktivacij, utežena strategija ravno tako kot na homogenem omrežju obnesla bolje, saj glede na vrednosti uteži računalniki, enaki strojni in programski opremi navkljub, niso popolnoma enako hitri. Razlog za te majhne razlike je v sistemskih procesih, ki na različnih računalnikih opravljajo različno delo.

Našo domnevo potrjuje primerjava utežene in krožne porazdeljevalne strategije pri preiskovanju mreže velikosti 32×32 stopnje 12: navkljub temu, da smo prostor vseh možnih postavitev razdelili na desetkrat večje podprostore s po 100000



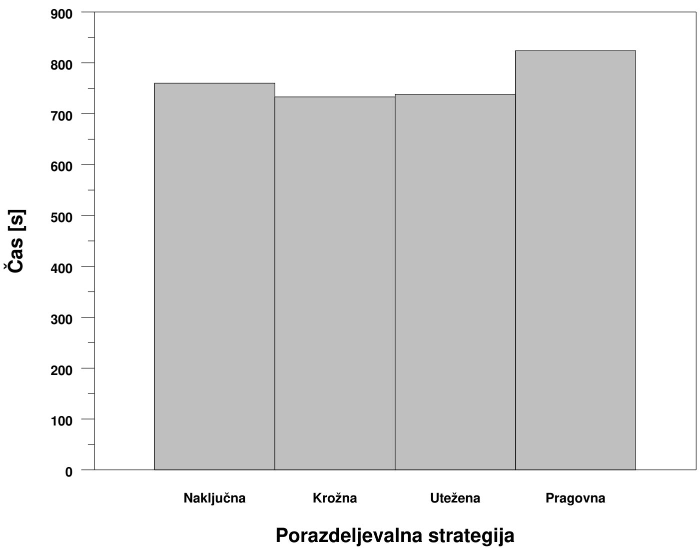
Slika 6.5: Čas računanja na homogenem omrežju (z uporabo predpomnjenja)



Slika 6.6: Primerjava utežene in krožne strategije na večjem problemu

postavitvami, je potrebnih 1464 aktivacij. Na sliki 6.6 vidimo, da se je utežena strategija pri večjem številu izvedenih aktivacij obnesla bolje.

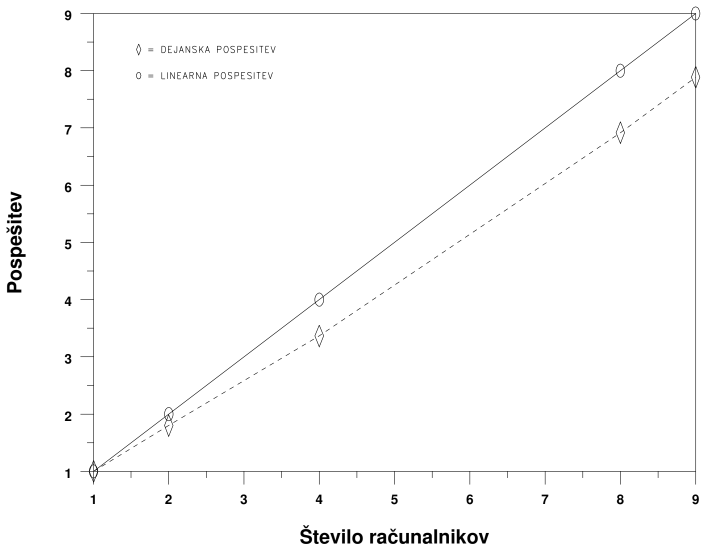
Oglejmo si še, kako se obnaša naša aplikacija, če ne uporabimo predpomnjenja referenc in seznamov strežnikov. Slika 6.7 pokaže, da vse porazdeljevalne strategije porabijo neprimerno več časa, kot pri uporabi predpomnjenja (slika 6.5), pa tudi bolj izenačene so. Analiza poteka izvajanja pokaže, da se sočasno uporabljajo le trije ali štirje računalniki, čeprav jih je na voljo devet. To je posledica tega, da je čas, potreben za preiskavo posameznega podprostora (okrog 9 sekund), v istem velikostnem razredu kot čas, potreben za aktivacijo (dobre 3 sekunde). Prvi aktivirani objekt konča svoje delo že, ko je aktiviran četrti. Na to težavo smo opozorili že pri preizkušanju predpomnjeja referenc v razdelku 6.3



Slika 6.7: Čas računanja na homogenem omrežju (brez uporabe predpomnjenja)

6.5 Pospešitev aplikacije

V homogenem omrežju lahko, spričo podobne računske moči vseh računalnikov, analiziramo tudi odvisnost med številom uporabljenih računalnikov in časom, potrebnim za izvajanje aplikacije. V idealnem primeru bi pričakovali linearno pospešitev: N računalnikov bi problem rešilo N-krat hitreje od enega samega. Slika 6.8 pokaže, da je pospešitev aplikacije res zelo blizu linearni: manjše odstopanje gre na račun časa, potrebnega za aktivacijo, ki je večji, če uporabimo več računalnikov.



Slika 6.8: Pospešitev aplikacije

7 Zaključek

V diplomskem delu smo opisali sistem za porazdeljevanje bremena v objektno usmerjenih aplikacijah, ki smo ga v ta namen tudi razvili. Predstavili smo osnovne pojme objektno usmerjenega programiranja, objektni model CORBA ter problem porazdeljevanja bremena in slednjega vključili v CORBA okolje. Izbrali smo primerno izvedbo CORBA infrastrukture, ORB ORBit in skladišče izvedb OAF, ki tvorita osnovo našega sistema. Postavili smo tudi zahteve, ki jih mora naš sistem izpolnjevati.

Odločili smo se za izvedbo porazdeljevanja na sistemskem nivoju, saj je to najlaže izvedljivo brez posegov v izvorno kodo ORBa. Porazdeljevanje je skoraj popolnoma skrito pred uporabnikom sistema: ta mora sistem le inicializirati in izbrati primerno porazdeljevalno strategijo. Navkljub temu pa mu je omogočen popoln nadzor nad porazdeljevanjem, če to želi. Porazdeljevanje lahko nadzoruje z uporabo lastnih porazdeljevalnikov, katerih izvedba je izjemno preprosta, ali pa z navajanjem omejitev, ki jim mora ustrezati aktivirani objekt in so uporabnikom OAF že znane. Pomembno je, da naša izvedba vzpodbuja ponovno uporabo že obstoječih izvedb objektov, saj lahko te uporabljamo brez posegov v njihovo izvorno kodo in celo brez ponovnega prevajanja. To lahko močno zmanjša čas, ki je potreben za razvoj obsežnih porazdeljenih aplikacij.

Programiranje na nivoju CORBE je, še posebej v jezikih, ki sintaktično ne podpirajo objektno usmerjenega programiranja, pogosto nerodno, zato smo razvili tudi knjižnico `libodf`, ki olajša uporabo sistema v jeziku C. Njen programski vmesnik je izjemno preprost in omogoča enostavno uporabo tako sistema ODF kot nekaterih lastnosti specifikacije CORBA, ki so nujne za vzporedno programiranje. Prednost uporabe jezika C je tudi v tem, da je knjižnico lahko uporabljati tudi v drugih programskih jezikih, saj večina teh podpira klicanje funkcij, napisanih v C.

Zavoljo časovno zahtevnega postopka aktivacije smo v sistem vključili tudi mehanizem predpomnjenja, ki močno skrajša čas aktivacij in prispeva k stabilnosti aplikacij.

Za preizkušanje sistema smo uporabili štiri preproste porazdeljevalne strategije: tri statične in eno dinamično. Slednja upošteva dejansko obremenitev računalnikov v času porazdeljevanja objektov, zato smo za njeno izvedbo razvili še objekt za merjenje obremenitve računalnikov, ki deluje na več različnih operacijskih sistemih.

Preizkusi so pokazali, da se razviti sistem ob uporabi primerne porazdeljevalne strategije dobro obnese tako na heterogenih kot na homogenih omrežjih:

na homogenih omrežjih, ki nam omogočajo neposredno preučevanje odvisnosti časa, porabljenega za računanje, od števila računalnikov, je pospešitev pri uporabi večjega števila računalnikov zelo blizu linearni.

7.1 Nadaljnje delo

Med pomembnejšimi nalogami je vsekakor izboljšanje varnosti sistema, saj ta zaradi pomanjkanja kakršnega koli avtentifikacijskega mehanizma ni primeren za uporabo v odprtih omrežjih.

Za uporabo sistema v omrežjih, v katerih se mešajo delovne postaje z operacijskimi sistemi Windows in UNIX, bi veljalo prenesti celotni sistem še na okolje Windows.

Mehanizem predpomnjenja bi bilo smotrno namesto na nivoju knjižnice `libodf` izvesti na nivoju objekta `Register`, kar bi omogočilo izkoriščanje tega mehanizma tudi pri uporabi sistema na nivoju CORBE in ne le preko knjižnice.

Sistem bi lahko izboljšali z izvedbo porazdeljevalnikov, ki bi uporabljali boljše porazdeljevalne strategije. Zanimivo bi bilo razviti tudi "meta-porazdeljevalnik", ki bi glede na lastnosti aplikacije in stanje omrežja za porazdeljevanje uporabljal različne "prave" porazdeljevalnike.

Literatura

[1] Object Management Group, Discussion of the Object Management Architecture, *http://www.omg.org/library/oma/oma-all.ps*, 1997

[2] Object Management Group, The Common Object Request Broker: Architecture and Specification - Revision 2.2, *ftp://ftp.omg.org/pub/docs/formal/98-07-01.ps*, 1998

[3] Henning, M., Vinoski, S., Advanced CORBA Programming with C++, *Addison Wesley Longman, Inc.*, 1999

[4] Rackl, G., Load Distribution for CORBA Environments, diplomsko delo, *Technische Universität München, Institut für Informatik*, 1997

[5] Shirazi, B. A., Hurson, A. R., Kavi, K. M., editors, *Scheduling and Load Balancing in Parallel and Distributed Systems*, IEEE Computer Society Press, 1995

[6] Schnekenburger T., *Automatic Load Distribution for CORBA Applications*, Component Users Conference, Munich, Germany, SIGS, ISBN 1-884842-57-7, 1996

[7] Močnik J., Trobec R., Robič B., Integration of Load Balancing in CORBA Environment, in *Proceedings of the International Workshop Parallel Numerics 2000, p. 229-234*, ISBN 80-968374-1-9

[8] Kodek, D., Mikroprocesorski sistemi, založba BI-TIM d.o.o., Ljubljana, 1993

[9] Vinoski, S., CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments, in IEEE Communications Magazine, Vol. 35, No.2, February 1997

[10] Schnekenburger T., Rackl G., Implementing Dynamic Load Distribution Strategies with Orbix, in International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '97), Las Vegas, Nevada, Volume II, p. 996-1005, CSREA, ISBN 0-9648666-6-8, 1997

[11] Azar, Y., Kalyanasudaram, B., Plotkin, S., Pruhs, K., Waarts, O., Online Load Balancing of Temporary Tasks, Workshop on Algorithms and Data Structures, p. 119-130, 1993

[12] Barth, T., Flender G., Freisleben B., Thilo F., Load Distribution in a CORBA Environment, in Proc. 1999 International Symposium on Distributed Objects and Applications (DOA'99), pp. 158-166, Edinburgh, Scotland, September 1999

[13] The GNOME Project Homepage, *http://www.gnome.org*

[14] ORBit Resources, *http://orbit-resources.sourceforge.net*

Stvarno kazalo

Zahvala

Za mentorstvo se zahvaljujem prof. dr. Borutu Robiču.

Dr. Roman Trobec je pomagal s številnimi vsebinskimi in oblikovnimi nasveti.

Računalnike, ki so tvorili obe preizkusni omrežji, je velikodušno dal na voljo odsek E6 Inštituta Jožef Stefan.

Dick Porter in Elliot Lee sta potrpežljivo odgovarjala na moja vprašanja o ORBitu in OAF ter mi razkrivala skrivnosti nekaterih delov izvorne kode obeh programov.

Uroš Jovanovič je napisal prvotno, nevzporedno različico preizkusne aplikacije.

Barbari Bizjak gre zahvala za lektoriranje diplomskega dela in neskončno potrpljenje med mojim pisanjem le-tega.

Tega dela bi ne bilo brez odličnih programskih orodij GNU C Compiler (*gcc*), GNU Source Level Debugger (*gdb*) in Emacs (*emacs*). Hvala vsem avtorjem!

Izjava

Izjavljam, da sem diplomsko delo izdelal samostojno pod vodstvom mentorja prof. dr. Boruta Robiča. Izkazano pomoč drugih sodelavcev sem v celoti navedel v zahvali.