

Aplicații Integrate pentru Întreprinderi

Laborator 6

08.11.2011

Dezvoltarea unei aplicații distribuite folosind tehnologia CORBA

Scopul laboratorului îl reprezintă înțelegerea mecanismului CORBA pentru dezvoltarea unei aplicații distribuite, prin apelul la distanță al metodelor unor obiecte rezidente pe server, funcționalitate de care clientul este informat prin specificația în cadrul unei interfețe independente de limbajul de programare.

1. CORBA în contextul aplicațiilor distribuite
2. Arhitectura CORBA
3. Specificații IDL
4. Etapele dezvoltării unei aplicații distribuite folosind CORBA
5. Studiu de Caz: Comparație între modelele de programare CORBA

1. CORBA în contextul aplicațiilor distribuite

CORBA (*eng* Common Object Request Broker Architecture), dezvoltată de OMG¹ (*eng* Object Management Group), este o tehnologie care oferă o arhitectură independentă de platformă și de limbajul de programare² pentru a dezvolta aplicații distribuite, orientate pe obiecte. Obiectele CORBA se pot afla oriunde, comunicând prin intermediul unei rețele de calculatoare, presupunând că mașinile pe care se găsesc sunt conectate la Internet, fiind vizibile în acest spațiu printr-o adresă publică. CORBA reprezintă un standard ce permite interacțiunea între (colecții de) obiecte distribuite (și eterogene) în cadrul unei aplicații.

Așadar, CORBA pune la dispoziție un mediu pentru dezvoltarea de aplicații distribuite, soluție adoptată atunci când datele folosite sunt distribuite³ (în baze / depozite de date aflate pe mașini diferite), când procesarea este distribuită⁴ (în sisteme de tip Grid / Cloud) sau când utilizatorii sunt distribuiți.

Caracteristicile sistemelor distribuite pot fi sistematizate astfel [1]:

	sisteme nedistribuite	sisteme distribuite
comunicație	rapidă	lentă
erori	obiectele sunt distruse împreună	obiectele sunt distruse separat
acces concurent	doar prin fire de execuție	da
securitate	da	nu

¹ Organizația Object Management Group, responsabilă pentru tehnologia CORBA, este formată din peste 300 de companii, cuprinzând majoritatea producătorilor de tehnologii distribuite (platforme, baze de date, aplicații, diferite utilitare). Mai mult informații pot fi obținute de la <http://www.omg.org>.

² Limbajul Java reprezintă o soluție pentru dezvoltarea de aplicații CORBA, argumente în acest sens fiind maparea cu limbajul IDL și sistemul de gestiune a memoriei (garbage collection).

³ Alte motive pentru distribuirea datelor ar putea fi probleme cu privire la: securitatea datelor, sau separarea datelor din rațiuni istorice (cronologice) sau al semnificației.

⁴ Se urmărește obținerea unui avantaj competitiv prin folosirea prelucrării paralele (acolo unde poate fi realizată o astfel de descompunere în activități care să fie executate concomitent). Totodată, specializarea unor mașini (servere pentru baze de date, sisteme grafice) poate reprezenta un motiv pentru a recurge la distribuirea unei aplicații.

Ținând cont de faptul că în cadrul aceluiași proces comunicația între obiecte este semnificativ mai rapidă decât în cazul unei aplicații distribuite, trebuie evitată o interacțiune foarte strânsă între obiecte în cadrul proiectării aplicațiilor ce utilizează prelucrarea paralelă.

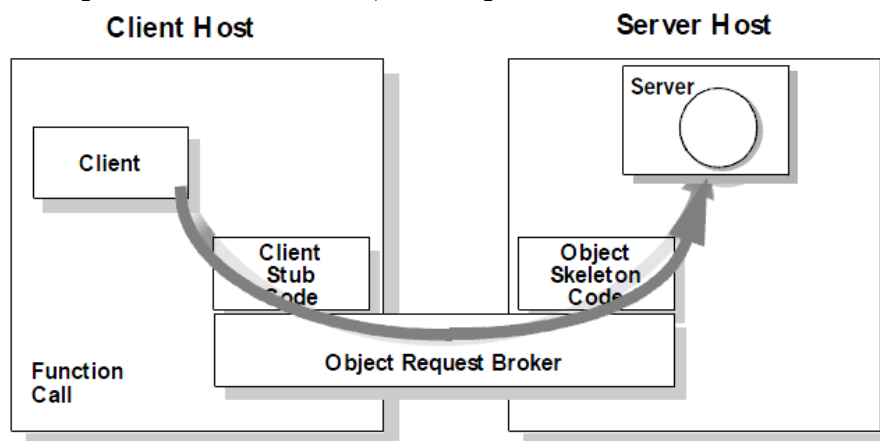
Într-un sistem tradițional, producerea unei erori implică distrugerea tuturor obiectelor create în cadrul procesului respectiv. Pe de altă parte, în cadrul aplicațiilor distribuite, programatorul trebuie să ia în calcul situația în care disponibilitatea obiectelor este diferită (existența lor fiind independentă), aceasta fiind dependentă de comportamentul procesului în care rulează.

Dacă pentru o aplicație nedistribuită există posibilitatea de a alege între unul sau mai multe fire de execuție, garantând un acces secvențial al obiectelor, respectiv implicând necesitatea folosirii unor mecanisme de sincronizare pentru controlul accesului concurent, în situația unui sistem ce rulează pe mai multe mașini, obiectele pot fi accesate de mai mulți clienți în mod implicit, astfel încât mecanismele de sincronizare trebuie implementate obligatoriu.

Securitatea este un aspect care trebuie luat în calcul atunci când obiectele sunt distribuite, interacțiunea între acestea trebuind să aibă la bază mecanisme de autentificare spre a se respecta drepturile de accesare ale acestora.

2. Arhitectura CORBA

CORBA definește o arhitectură pentru obiecte distribuite, bazată pe conceptul de cerere a unui serviciu pus la dispoziție de un obiect distribuit. Serviciile pe care le oferă un obiect distribuit sunt specificate în interfața sa, descrisă într-un limbaj de definire al interfeței dezvoltat de OMG (IDL – *eng.* Interface Definition Language). Obiectele vor fi identificate prin referințele la ele, având tipurile specificate în interfața corespunzătoare.



Invocarea unui obiect distribuit CORBA [2]

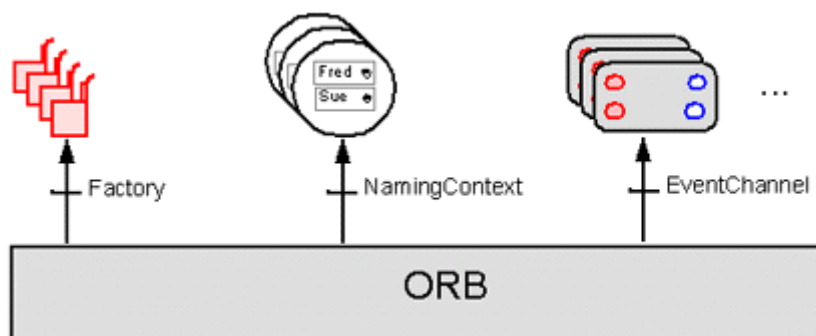
Clientul deține o referință către un obiect distribuit (rezident pe server), definit printr-o interfață. Serviciul (distribuit) ORB (*eng.* Object Request Broker) transmite cererea obiectului distribuit pe server și întoarce rezultatul la client. Pentru a implementa acest comportament, serviciul localizează obiectul în rețea, comunică cererea către acesta, așteaptă rezultatul, pentru ca în momentul în care este disponibil să îl transmită către aplicația care l-a solicitat. Așadar, ORB realizează transparența atât față de **locația** unde se află implementat obiectul care pune la dispoziție serviciile cât și față de **limbajul de programare** în care se scrie cererea (acesta putând fi diferit față de limbajul de programare în care este implementat serviciul obiectului), realizând corespondența necesară.

Specificația CORBA are drept obiectiv portabilitatea atât a clientului cât și a implementării obiectelor. În acest sens, este definit un API atât pentru clienții unui obiect distribuit cât și pentru implementarea obiectului CORBA⁵.

Pentru asigurarea interoperabilității, este definit un protocol IIOP (*eng.* Internet Inter-ORB Protocol) ce permite clienților folosind un produs CORBA realizat de un producător să poată comunica cu obiecte din cadrul altui produs CORBA dezvoltat de orice alt producător. IIOP funcționează peste orice implementare TCP/IP.

În cadrul unui sistem distribuit, interoperabilitatea este mai importantă decât portabilitatea. Astfel IIOP este folosit și în alte sisteme decât în cele care implementează API-ul CORBA⁶. Deoarece toate aceste sisteme folosesc IIOP ca protocol de transport, deși utilizează API-uri diferite, pot interacționa între ele.

Un aspect important în arhitectura CORBA este reprezentat de definirea unui set de servicii distribuite⁷ care suportă integrarea și interacțiunea dintre obiecte distribuite. Ele sunt implementate ca obiecte distribuite CORBA, definite prin interfețe IDL.



Servicii CORBA [1]

Serviciu	Descriere
ciclu de viață al obiectului	definește procesele de creare, ștergere, mutare și copiere a obiectelor CORBA
serviciu de nume	definește modul în care se pot asocia obiectelor CORBA nume simbolice
serviciu de evenimente	decuplează comunicarea între obiecte CORBA
serviciu de legături	stabilește legături (cu multiplicități și tipuri asociate) între obiecte CORBA
serviciu de externalizare	realizează transformarea obiectelor CORBA în / din surse externe

⁵ Acest obiectiv ar presupune că se poate rescrie cu ușurință codul dezvoltat pentru accesarea unui serviciu CORBA al unui producător, astfel încât să poată funcționa și în alte contexte similare. În realitate, aplicații de tip client sunt portabile, însă adaptarea în cazul implementărilor obiectelor este mai dificilă pentru a asigura interoperabilitatea cu mai multe produse CORBA.

⁶ IIOP este folosit ca protocol de transport pentru o versiune a Java RMI (RMI peste IIOP). Enterprise Java Beans pot folosi IIOP (deoarece sunt definite prin intermediul RMI). Totodată, numeroase servere de aplicații utilizează IIOP fără a implementa întreg API-ul CORBA.

⁷ Acestea sunt cunoscute sub numele de COS (*eng.* CORBA Services). Câteodată sunt referite ca Object Services.

serviciu de tranzacții	coordonează atomicitatea accesului la obiecte CORBA
control al accesului concurent	oferă servicii de blocare a accesului la obiecte CORBA pentru asigurarea mecanismului de serializare
serviciu de proprietăți	stabilește asocierea perechilor de tip nume-valoare cu obiecte CORBA
serviciu de identificare	găsește obiecte CORBA pe baza proprietăților care descriu serviciul oferit
serviciu de interogări	permite realizarea de interogări cu obiecte CORBA

CORBA este o specificație implementată de mai mulți producători pentru diferite limbaje de programare. Pentru Java, au fost dezvoltate Java ORB (distribuită împreună cu SDK-ul Java, având unele funcționalități lipsă), VisiBroker for Java (dezvoltată de Borland), Orbix Web (de la Iona Technologies), WebSphere (server de aplicații realizat de IBM).



Exemplu

În cele ce urmează, implementarea unei aplicații distribuite folosind tehnologia CORBA va fi ilustrată pe cazul particular al aplicației implementând funcționalitatea serviciului Doodle, al cărui comportament a fost descris pe larg în laboratorul anterior.

3. Specificații IDL

Prin limbajul de definire al interfețelor dezvoltat în cadrul OMG (IDL – *eng.* Interface Definition Language) se permite specificarea obiectelor distribuite prin operațiile suportate, fără a se oferi detalii cu privire la modul în care acestea sunt realizate⁸. Implementarea unui obiect CORBA este realizată într-un limbaj de programare, interfața fiind doar o punte de legătură (un contract⁹) între codul care folosește obiectul¹⁰ și codul care definește obiectul.

Interfața în care este specificat comportamentul obiectului distribuit este independentă de limbajul de programare. IDL definește corespondențe pentru mai multe limbaje de programare¹¹, permițând deci ca implementarea unui obiect să se poată face în limbajul de programare adecvat pentru obiectul respectiv. Totodată, același lucru este posibil și pentru client.

Pot fi definite astfel, independent de limbajul de programare, interfețe modularizate pentru obiecte, specificând atribute și metode suportate de acestea. De asemenea, pot fi indicate excepțiile pe care le poate genera o anumită operație. Fiecare operație este definită prin tipul de date al valorii întoarse precum și prin tipurile de date ale parametrilor.

IDL suportă tipuri de date de bază (*boolean, char, octet, short, long, float, double, string*) sau construite (*struct, union, enum, sequence*), obiecte definite sau tipul *any* pentru obiecte cu tipul de date stabilit dinamic (la momentul utilizării).

⁸ Prin urmare, nu se vor specifica în IDL stări ale obiectului sau algoritmi.

⁹ Interfața poate fi considerată un contract în sensul garantării unei anumite funcționalități (prin parametrii de intrare și de ieșire) fără a oferi detalii despre modul de implementare.

¹⁰ Clientul este dependent doar de interfață.

¹¹ Au fost standardizate corespondențe între IDL și C, C++, Java, Ada, COBOL, SmallTalk, Objective C, Lisp.

Correspondența între tipuri de date IDL și cele din limbajul de programare Java sunt descrise mai jos:

IDL	Java
boolean	boolean
char / wchar	char
octet	byte
short / unsigned short	short
long / unsigned long	int
long long / unsigned long long	long
float	float
double	double
string / wstring	String

De asemenea, există o corespondență între anumite structuri IDL și structuri Java.

IDL	Java
module	package
interface	interface
operation	method
attribute	pair of methods
exception	exception

Un concept întâlnit frecvent atunci când se vorbește despre corespondența tipurilor de date este cel de tipare inversă (*eng.* marshaling), adică de transformare a unei structuri specifice unui limbaj de programare într-un format CORBA IIOP, astfel încât să poată fi transmisă prin rețea.

Pentru aplicația considerată, un exemplu de specificație IDL poate fi:

```
module Programare
{
    struct Data
    {
        long zi;
        long luna;
        long an;
    };
    struct Moment
    {
        long ora;
        long minut;
    };
    struct Timp
    {
        Data d;
        Moment m;
    };
    struct Interval
    {
        Timp inceput;
        Timp sfarsit;
    };

    typedef sequence<Interval> Intervale;
}
```

```

exception OrarNespecificat {};

interface ServiciuProgramare {
    Intervale obtineOrar() raises(OrarNespecificat);
    Intervale obtineIntervaleOcupate() raises(OrarNespecificat);
    Intervale obtineIntervaleLibere() raises(OrarNespecificat);
    boolean ocupaInterval(in long identificador, in Interval interval)
raises(OrarNespecificat);
    boolean elibereazaInterval(in long utilizator, in Interval interval)
raises(OrarNespecificat);
};
};

```

În specificația IDL `Programare.idl`, au fost definite ca structuri de date `Data`, `Moment`, `Timp`, `Interval`, `Intervale`, și excepția `OrarNespecificat`, utilizate ulterior în interfața `ServiciuProgramare`. Au fost definite mai multe metode care primesc parametri de intrare (fapt specificat prin cuvântul cheie `in`¹²), descriind funcționalitatea oferită de aplicație.

Produsele CORBA pun la dispoziție un compilator care convertește interfața IDL în reprezentarea corespunzătoare limbajului de programare. Pentru distribuția standard Java, acesta se numește `idlj`¹³.

Prin comanda

```
> idlj Programare.idl
vor fi generate următoarele fișiere14:
```

Fișier	Semnificație
<code>ServiciuProgramare.java</code>	Interfața IDL reprezentată ca interfață Java
<code>ServiciuProgramareHelper.java</code>	Implementează operațiile de tipare pentru interfață (<code>insert</code> , <code>extract</code> , <code>read</code> , <code>write</code> , <code>narrow</code> , <code>unchecked_narrow</code>)
<code>ServiciuProgramareHolder.java</code>	E folosit pentru operații cu parametri de tip <code>out</code> și <code>inout</code> .
<code>ServiciuProgramareOperations.java</code>	Conține descrierile operațiilor implementate de interfață (<code>ServiciuProgramare</code> este derivată din <code>ServiciuProgramareOperations</code>)
<code>_ServiciuProgramareStub.java</code>	Implementează un obiect local reprezentând obiectul CORBA „la distanță” ce transmite cererile spre obiectul distribuit

Pentru a genera atât fișierele pentru client și pentru server, utilitarul `idlj` este apelat cu parametrul `-fall`. Pentru a genera doar fișierele pentru server sau pentru client sunt folosiți parametrii `-fserver`, respectiv `-fclient`¹⁵.

¹² Parametrii de ieșire (indicând date care sunt transmise de la obiectul distribuit către client) sunt specificați prin cuvântul cheie `out`, iar parametrii de intrare ieșire (indicând date transmise de la client către obiectul distribuit și înapoi la client) sunt specificați prin cuvântul cheie `inout`.

¹³ Utilitarul `idlj` se găsește în directorul `bin` al distribuției standard.

¹⁴ Sunt menționate doar fișierele cu semnificație în dezvoltarea aplicației distribuite, fiind omise fișierele corespunzătoare celorlalte structuri de date, al căror format (ca denumire și funcționalitate) este asemănător.

¹⁵ Implicite (dacă utilitarul este apelat fără parametri), sunt generate doar fișierele pentru client.

4. Etapele dezvoltării unei aplicații distribuite folosind CORBA

Așadar, etapele dezvoltării unei aplicații distribuite folosind CORBA sunt:

- definirea interfeței care stabilește funcționalitatea obiectului distribuit, folosind limbajul de specificare IDL, astfel încât programatorii să poată implementa programe client și server în orice limbaj de programare compatibil cu CORBA (când se implementează un server sau un client, interfețele IDL sunt primite de la producător);
- compilarea interfeței care conține funcționalitatea obiectului distribuit (pentru Java, se folosește `idlj`); se obține versiunea interfeței corespunzătoare limbajului de programare respectiv, și totodată clasele ciot (*eng. stub*) respectiv schelet (*eng. skeleton*) care conțin infrastructura de conectare la ORB;
- dezvoltarea serverului (se folosesc clasele schelet generate); pe lângă implementarea metodelor specificate în interfață, trebuie precizat mecanismul de pornire a serviciului ORB și de așteptare a invocărilor realizate de client;
- dezvoltarea clientului (se folosesc clasele ciot generate); clientul pornește serviciul ORB, caută serverul folosind serviciul de nume oferit de interfață pentru a obține o referință către obiectul distribuit, spre a-i apela metodele
- pornirea aplicațiilor în următoarea ordine: serviciu de nume, server, client. Prin serviciul de nume (referințele către) obiectele distribuite sunt publicate de server, devenind astfel disponibile clienților, care le pot folosi pentru a apela metodele specificate în interfață. Vom folosi `orbd`, un proces care rulează în fundal (*eng. daemon*) și conține serviciile: de inițializare (Bootstrap Service), de nume tranzitoriu¹⁶ (Transient Name Service) dar și de nume permanent (Persistent Name Service) și un proces de gestiune pentru server (Server Manager).

⚠ Într-un sistem Unix, serviciul de nume se pornește astfel:

```
# orbd -ORBInitialPort 1100&
```

🌈 Într-un sistem Windows, serviciul de nume se pornește astfel:

```
> start orbd -ORBInitialPort 1100
```

Parametrul `ORBInitialPort` este obligatoriu.

CORBA implementează un sistem de excepții care este foarte similar celui din limbajul de programare Java.

Există două tipuri de excepții în CORBA: **sistem** și **utilizator**.

Excepțiile de tip sistem sunt generate atunci când au loc evenimente ce țin de aplicație la modul general (a fost apelată o metodă care nu este implementată pe server, există o problemă de comunicare, sistemul ORB nu a fost inițializat corespunzător). *System Exceptions* în Java sunt o subclasă a `RuntimeException` astfel încât ele nu trebuie incluse într-un bloc `try {...} catch`. Totuși, tratarea excepțiilor în acest fel poate beneficia de avantajul obținerii unui cod de eroare, care diferă de la producător la producător.

Excepțiile de tip utilizator sunt generate în cazul unor erori în cadrul unei metode propriu-zise. *User Exceptions* în Java sunt o subclasă a `Exception`, așadar trebuie incluse (obligatoriu) într-un bloc `try {...} catch`.

¹⁶ Se poate folosi serviciul de nume (tranzitoriu) `tnameserv` (apelat fără parametri), ca alternativă la `orbd`. De regulă, `tnameserv` se lansează pe portul 900.

4.1 Proiectarea serverului

Implementarea obiectului distribuit CORBA este transparentă pentru client, care cunoaște doar specificațiile din interfața IDL¹⁷. La nivel de server, implementarea obiectului distribuit se face pe baza clasei schelet generată la compilarea interfeței IDL. Se oferă astfel mecanismul de acces la nivelul obiectului distribuit, ce conține despachetarea datelor primite, apelarea metodei ce implementează operația solicitată, împachetarea rezultatelor transmise.

Pentru implementarea obiectului distribuit CORBA, programatorul trebuie să dezvolte o clasă derivată din casa schelet, specificând comportamentul pentru metodele descrise de interfața IDL.

CORBA specifică două mecanisme de dezvoltare a serverului pentru implementarea unei interfețe IDL:

- modelul moștenirii, care presupune implementarea unei clase care extinde clasa schelet generată de compilator; din acest model fac parte:
 - standardul POA¹⁸ (Portable Object Adapter), dezvoltat de OMG; compilatorul generează o clasă `*POA.java` care extinde `org.omg.PortableServer.Servant`, folosită drept clasă de bază pentru toate implementările servantului (clasa care implementează metodele specificate în interfața IDL pe server);
 - `ImplBase` – este folosit doar pentru a asigura compatibilitatea cu servere scrise în versiuni Java mai vechi (< 1.4); este recomandată evitarea acestui model în favoarea POA.
- modelul delegării – interfața IDL este implementată folosind două clase
 - o clasă generată de compilator care moștenește clasa schelet, dar delegă toate apelurile unei clase care implementează metodele;
 - o clasă care implementează operațiile generate din interfața IDL (având denumirea `*Operations.java`).

La nivelul serverului, trebuie realizate următoarele operații:

- crearea și inițializarea instanței ORB;
- obținerea unei referințe către obiectul rădăcină POA și activarea `POAManager`;
- instanțierea servantului (implementarea obiectului distribuit);
- obținerea unei referințe către contextul de nume în care se va înregistra obiectul distribuit (obținerea obiectului rădăcină al contextului de nume);
- înregistrarea obiectului distribuit în contextul de nume;
- așteptarea invocărilor obiectului distribuit de către client.

¹⁷ Încapsularea completă a obiectelor distribuite CORBA oferă mai multă libertate în ceea ce privește implementarea lor, aceasta putând fi realizată în limbaje de programare diferite chiar și față de limbajul de programare în care este scris clientul.

¹⁸ Un obiect de adaptare (*eng.* object adapter) este un mecanism care conectează o cerere la codul care o deservește prin intermediul unui obiect referință. POA este un tip particular al acestui tip de obiect definit de specificația CORBA care îndeplinește următoarele obiective: permite programatorilor să dezvolte implementări de obiecte care sunt portabile între diferite produse ORB, oferă suport pentru obiecte persistente (din perspectiva clientului, obiectele există pe tot parcursul aplicației menținând valorile stocate în ele – deși serverul poate fi repornit sau implementările pot fi realizate prin obiecte diferite), pentru activarea transparentă a obiectelor, permițând ca un servant să suporte simultan mai multe identități de obiecte. Prin POA, implementarea deține controlul asupra identității, stării, stocării, ciclului de viață al obiectului.


```
import org.omg.CORBA.*;
import org.omg.CosNaming.*;
import org.omg.PortableServer.*;
import Programare.*;

public class Server
{
    public static void main (String[] args)
    {
        try
        {
            ORB orb = ORB.init(args,null);

            POA radacinapoa =
            POAHelper.narrow
            (orb.resolve_initial_references("RootPOA"));
            radacinapoa.the_POAManager().activate();

            ImplementareServiciuProgramare
            serviciuprogramareImpl = new ImplementareServiciuProgramare();
            org.omg.CORBA.Object
            serviciuprogramareImplRef =
            radacinapoa.servant_to_reference(serviciuprogramareImpl);
            ServiciuProgramare
            serviciuprogramareRef =
            ServiciuProgramareHelper.narrow(serviciuprogramareImplRef);

            org.omg.CORBA.Object
            serviciunumeRef =
            orb.resolve_initial_references("NameService");
            NamingContextExt
            contextnumeRef =
            NamingContextExtHelper.narrow(serviciunumeRef);

            String numeserviciu = "ServiciuProgramare";
            NameComponent
            componentanume = new NameComponent(numeserviciu,"");
            NameComponent cale[] = {componentanume};
            contextnumeRef.rebind(cale,serviciuprogramareRef);

            orb.run();
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Clasa `ImplementareServiciuProgramare` reprezintă servantul, care extinde clasa `ServiciuProgramarePOA`, astfel încât moștenește funcționalitatea CORBA generată de compilatorul IDL. Aceasta va conține toate implementările descrise prin interfață, conținând serviciile puse la dispoziție.

Un server CORBA are nevoie de un obiect ORB local, ca de altfel și fiecare client CORBA. Toate obiectele servanț vor fi înregistrate prin acesta, astfel încât serviciul ORB să poată localiza serverul atunci când sunt invocate funcționalități pe care le pune la dispoziție. Prin metoda `init()`, ce primește argumente din linia de comandă, se pot specifica anumite proprietăți, între care cele mai importante sunt portul și adresa pe care rulează serverul.

Serverul va fi lansat în execuție astfel:

```
# java Server -ORBInitialPort 1100 -ORBInitialHost localhost&  
> start java Server -ORBInitialPort 1100 -ORBInitialHost localhost
```

Apoi, prin intermediul serviciului ORB se obține obiectul care conține referința către obiectul POA rădăcină prin metoda `resolve_initial_references`. După obținerea referinței către obiectul POA rădăcină se activează serviciul `POAManager`. Prin metoda `activate()` se schimbă starea serviciului de gestiune POA în activ astfel încât obiectele asociate acestuia¹⁹ vor prelucra cererile venite de la client.

Serverul este procesul care instanțiază unul sau mai multe obiecte servant. Un servant moștenește interfața generată de compilatorul IDL, implementând propriu-zis prelucrările specificate prin operațiile respective. Servantul `ImplementareServiciuProgramare` este instanțiat prin obiectul `serviciuprogramareImpl`.

Se obține o referință către obiectul distribuit, inițial ca obiect CORBA (prin metoda `servant_to_reference()`, apelată din contextul obiectului rădăcină POA, instanței servantului), cast-ul la tipul corespunzător făcându-se prin metoda (statică) `narrow` din clasa (generată de compilatorul IDL) `ServiciuProgramareHelper`.

Operațiile puse la dispoziție de (instanța) servant(ului) sunt făcute vizibile către client prin serviciul de nume Common Object Service (COS)²⁰. Se obține referința către acest serviciu de nume astfel încât să se poată înregistra prin ea referințele către obiectele care implementează diferite interfețe²¹. Referința către serviciul de nume se face tot prin metoda `resolve_initial_references`, transformarea către tipul corespunzător contextului de nume realizându-se prin metoda `narrow`, la fel ca în cazul obiectelor POA. Parametrul "NameService" este definit pentru toate serviciile ORB, indicând faptul că serviciul de nume va fi persistent, dacă se folosește serviciul de nume `orbd`, respectiv tranzitoriu, în cazul utilizării `tnameserv`.

Se realizează o componentă de nume indicându-se denumirea prin care serviciile puse la dispoziție de obiectul respectiv pot fi accesate. Se face apoi legătura dintre componenta de nume și referința către obiectul distribuit, întoarsă către client prin apeluri tip `NamingContext.resolve_str(numeserviciu)`, prin care se poate obține acces către funcționalitatea dorită.

Apelul `orb.run()` blochează serverul în așteptarea invocărilor realizate din contextul clientului. Deoarece apelul este făcut din metoda `main`, este folosit firul de execuție corespunzător acesteia, astfel încât, la momentul încheierii prelucrărilor corespunzătoare unei invocări, serverul va rămâne în continuare blocat în așteptarea unor alte apeluri²².

¹⁹ Fiecare obiect POA are asociat un obiect `POAManager` (care poate gestiona, la rândul lui, unul sau mai multe obiecte POA). Prin `POAManager` se încapsulează starea în care se găsește prelucrarea fiecărui obiect POA asociat.

²⁰ Obiectele pot fi făcute vizibile către client și prin transformarea într-un șir de caractere a referințelor către acestea și publicarea lor într-un fișier.

²¹ Referințele către obiectele care implementează diferite interfețe sunt folosite de clienți pentru invocarea metodelor specificate în interfață.

²² Datorită acestui comportament, trebuie specificată o condiție de terminare, implicând apelul metodei `shutdown()` asociată obiectului corespunzător serviciului ORB.

4.2 Proiectarea clientului

Operațiile realizate la nivelul clientului sunt asemănătoare cu cele realizate de către server:

- crearea și inițializarea instanței ORB – se folosește metoda `init()` la fel ca în cazul serverului, iar parametrii (din linia de comandă) indică, similar, portul și adresa unde rulează serviciul de nume;

```
# java Client -ORBInitialPort 1100 -ORBInitialHost localhost  
> java Client -ORBInitialPort 1100 -ORBInitialHost localhost
```

Orice client are nevoie de un obiect local ORB pentru realizarea împachetării datelor transmise și prelucrărilor IIOP.

- obținerea unei referințe către contextul de nume folosit pentru identificarea (unei referințe către) obiectul(ui) distribuit – este apelată metoda `resolve_initial_references` din contextul obiectului ORB având parametrul "NameService", cu aceeași semnificație ca în cazul serverului, transformarea la tipul dorit realizându-se prin metoda `narrow`;
- utilizarea serviciului de nume pentru căutarea obiectului distribuit prin specificarea unei denumiri sub care aceasta a fost înregistrat (de către serverul pe care se găsește o implementare a sa) – se folosește metoda `resolve_str` a obiectului care conține o referință contextul de nume, având ca parametru denumirea sub care a fost specificat, în același mod precum s-a realizat localizarea serviciului de nume;
- transformarea (referinței către) obiectul(ui) distribuit CORBA obținut la tipul corespunzător folosindu-se metoda `narrow`;
- apeluri ale metodelor puse la dispoziție de obiectul distribuit CORBA, specificate în interfața IDL – invocările CORBA arată la fel ca apelul unei metode realizată asupra unui obiect local, detaliile cu privire la împachetare și transmitere fiind transparente către utilizator.

```
import org.omg.CORBA.*;  
import org.omg.CosNaming.*;  
import Programare.*;  
public class Client  
{  
    public static void main (String[] args)  
    {  
        try  
        {  
            ORB orb = ORB.init(args,null);  
            org.omg.CORBA.Object  
            serviciunumeRef =  
            orb.resolve_initial_references("NameService");  
            NamingContextExt  
            contextnumeRef =  
            NamingContextExtHelper.narrow(serviciunumeRef);  
            String numeserviciu = "ServiciuProgramare";  
            ServiciuProgramare  
            serviciuprogramareRef =  
            ServiciuProgramareHelper.  
            narrow(contextnumeRef.resolve_str(numeserviciu));  
            // TO DO: apeleaza servicii obiect programare  
        }  
        catch (Exception e) { e.printStackTrace(); }  
    }  
}
```

5. Studiu de Caz: Comparație între modelele de programare CORBA implementate în Java

IIOP este un protocol de transport (peste Internet) între obiecte tip ORB având drept scop realizarea interoperabilității între limbaje de programare și aplicații comercializate de diverși producători. Acest protocol este utilizat pentru sisteme distribuite dezvoltate fie în Java RMI, fie în IDL.

Modelul de programare IDL este axat pe interfața care descrie metodele care pot fi apelate din procesul „la distanță”, specificând tipurile de argumente acceptate de procedura invocată cât și tipurile informațiilor întoarse de aceasta. CORBA este un sistem independent de limbajul de programare în care valorile argumentelor sau rezultatelor întoarse sunt limitate la ceea ce poate fi reprezentat în limbajele de programare în care se realizează implementarea²³. Totodată, orientarea pe obiecte este limitată la obiecte transmise prin referință (obiectul nu poate fi transmis între mașini prin cod).

Modelul de programare IDL (Java™ IDL) conține Java CORBA ORB și compilatorul `idlj` – care realizează corespondența între limbajele IDL și Java. Astfel, este implementată funcționalitatea CORBA în platforma Java, oferind interoperabilitate și conectivitate standardizată permițând aplicațiilor distribuite să invoce – în mod transparent – operații puse la dispoziție prin rețea folosind IDL și IIOP. Este folosit un obiect ORB pentru prelucrări distribuite folosind comunicație bazată pe IIOP. Folosirea acestui model implică definirea interfețelor „la distanță” folosind limbajul IDL (dezvoltat de OMG), compilarea lor folosind `idlj` care va genera versiunea Java a interfeței ca și clasele ciot și schelet care permit aplicațiilor să comunice prin ORB. Java IDL este o componentă a distribuției standard Java.

Limbajul IDL de specificare a interfețelor este pur declarativ, proiectat pentru specificarea operațiilor corespunzătoare unei aplicații distribuite independent de limbajul de programare. El cuprinde și corespondențele către limbaje de programare ca Java, C, C++, Lisp, Python, Smalltalk, COBOL și Ada, astfel că fiecare instrucțiune este translatată corespunzător. Astfel, un client dezvoltat în Java se poate folosi de implementările (specificate ca interfață IDL) realizate în C++ pentru un anumit serviciu, interoperabilitatea dintre cele două limbaje de programare fiind obținută prin intermediul ORB.

În RMI, interfața și implementarea ei sunt dezvoltate folosind același limbaj de programare, astfel încât nu există problema corespondenței dintre acestea. Obiecte la nivel de limbaj de programare (codul însuși) pot fi transmise de la un proces la altul.

Modelul de programare RMI se referă la prelucrări distribuite prin API-ul cu același nume. Se poate lucra doar în limbajul de programare Java, folosind Java Remote Method Protocol (JRMP) sau cu limbaje de programare compatibile CORBA, folosind protocolul IIOP. Acest model de programare este parte din distribuția standard Java și conține serviciul ORB și compilatorul `rmi_c` pentru generarea claselor ciot și schelet precum și a legăturilor dintre obiecte la distanță folosind protocolele JRMP sau IIOP. Folosind același limbaj de programare, dezvoltarea este mai rapidă, iar flexibilitatea este obținută prin posibilitatea transmiterii de obiecte serializabile.

²³ Tipurile parametrilor de intrare și de ieșire trebuie să fie cele specificate în interfață.

Modelul de programare IDL²⁴ sau RMI cu componenta IIOP vor fi folosite așadar doar pentru interfațarea cu anumite aplicații (*eng.* legacy systems), preferându-se folosirea modelului de programare RMI din motive ce țin de portabilitate, securitate și colectarea memoriei disponibile.



Activitate de Laborator

[0p] 1. Să se pornească serviciul de nume orbd prin apelarea scripturilor `pornesteserviciudenum` `[.bat|.sh]`.

[0p] 2. Să se compileze interfața `Programare.idl` folosind compilatorul `idlj` pentru generarea claselor `server`, respectiv `client`.

```
idlj -fserver Programare.idl  
idlj -fclient Programare.idl
```

Clasele generate au fost deja incluse în proiectele Eclipse / NetBeans, rularea comenzii având scop didactic, spre a observa mecanismul de generare a claselor precum și conținutul lor.

[1p] 3. Să se implementeze un constructor al clasei

`ImplementareServiciuProgramare` care să permită specificarea unui orar.

În acest sens, ar putea fi utilă implementarea unei metode de conversie între clasa Java `Calendar` și clasa `Time`, specificată în interfața IDL.

[1p] 4. Să se implementeze metoda `obtineOrar` a clasei

`ImplementareServiciuProgramare`.

[1p] 5. Să se implementeze metoda `obtineIntervaleOcupate` a clasei

`ImplementareServiciuProgramare`.

[3p] 6. Să se implementeze metoda `obtineIntervaleLibere` a clasei

`ImplementareServiciuProgramare`.

[2p] 7. Să se implementeze metoda `ocupaInterval` a clasei

`ImplementareServiciuProgramare`.

[2p] 8. Să se implementeze metoda `elibereazaInterval` a clasei

`ImplementareServiciuProgramare`.

[0p] 9. Să se testeze că metodele `ocupaInterval` și `elibereazaInterval` au fost implementate corect.

BONUS.

[1p] 10. Să se modifice comportamentul metodei `elibereazaInterval` astfel încât să fie permisă eliberarea unei porțiuni din intervalul ocupat inițial.

Bibliografie

[1] Introduction to CORBA

<http://java.sun.com/developer/onlineTraining/corba/corba.html>

[2] *Orbix, CORBA Programmer's Guide, Java (version 6.3)*, Iona Technologies, 2006

²⁴ Folosirea acestui model de programare nu exclude posibilitatea utilizării exclusive a limbajului de programare Java pentru a beneficia de avantajul portabilității și a robusteții platformei pusă la dispoziție de acesta.