Aplicații Integrate pentru Întreprinderi

Laborator 6

11.11.2013

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

CORBA (*eng* Common Object Request Broker Architecture), dezvoltată de OMG[1] (*eng* Object Management Group), este o tehnologie care oferă o arhitectură independentă de platformă şi de limbajul de programare[2] pentru a dezvolta aplicații distribuite, orientate pe obiecte. Obiectele CORBA se pot afla oriunde, comunicând prin intermediul unei rețele de calculatoare, presupunând că maşinile 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 (şi eterogene) în cadrul unei aplicații.

Aşadar, CORBA pune la dispoziție un mediu pentru dezvoltarea de aplicații distribuite, soluție adoptată atunci când datele folosite sunt distribuite[3] (în baze / depozite de date aflate pe maşini diferite), când procesarea este distribuită[4] (în sisteme de tip Grid / Cloud) sau când utilizatorii sunt distribuiți.

Caracteristicile sistemelor distribuite pot fi sistematizate astfel [1]:

Criteriu	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 şi 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 maşini (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 aceluiaşi 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.

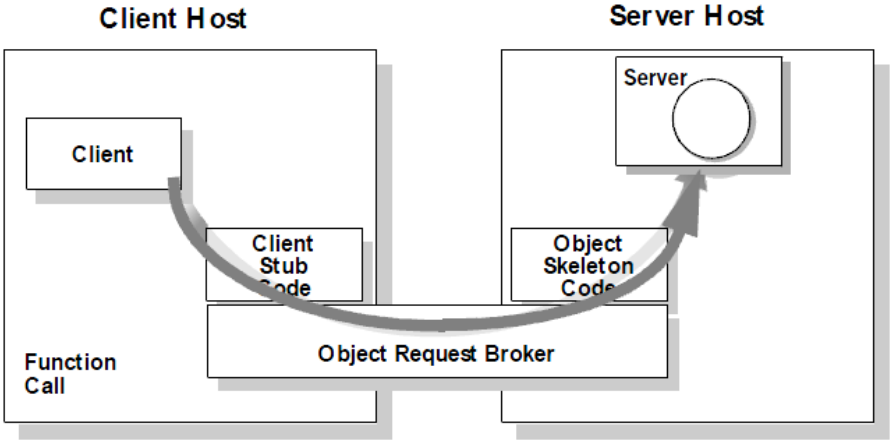
Într-un sistem tradițional, producerea unei erori implică distrugerea tuturor obiectelor create în cadrul procesului respectiv. Pe de altă parte, în cadrul aplicațiilor distribuite, programatorul trebuie să ia în calcul situația în care disponibilitatea obiectelor este diferită (existența lor fiind independentă),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

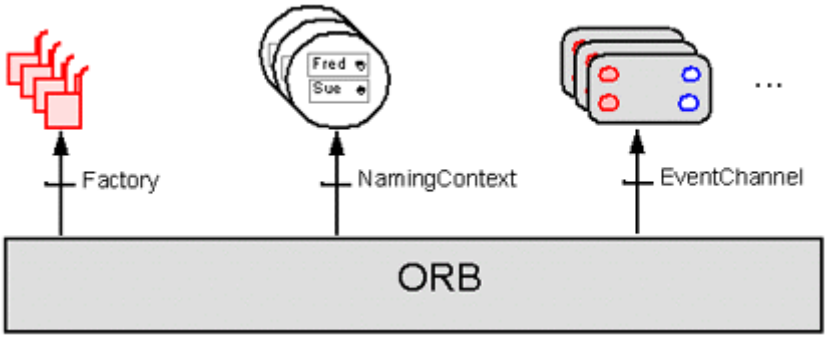
Clientul deține o referință către un obiect distribuit (rezident pe server), definit printr-o interfață. Serviciul (distribuit) ORB (*eng.* Object Request Broker) transmite cererea obiectului distribuit pe server şi întoarce rezultatul la client. Pentru a implementa acest comportament, serviciul localizează obiectul în rețea, comunică cererea către acesta, aşteptă rezultatul, pentru ca în momentul în care este disponibil să îl transmită către aplicația care l-a solicitat. Aşadar, ORB realizează transparența atât față de **locația** unde se află implementat obiectul care pune la dispoziție serviciile cât şi 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 şi a implementării obiectelor. În acest sens, este definit un API atât pentru clienții unui obiect distribuit cât şi pentru implementarea obiectului CORBA[5].

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 şi în alte sisteme decât în cele care implementează API-ul CORBA[6]. Deoarece toate aceste sisteme folosesc IIOP ca protocol de transport, deşi utilizează API-uri diferite, pot interacționa între ele.

Un aspect important în arhitectura CORBA este reprezentat de definirea unui set de servicii distribuite[7] care suportă integrarea şi 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	defineşte procesele de creare, ştergere, mutare şi copiere a obiectelor CORBA
serviciu de nume	defineşte modul în care se pot asocia obiectelor CORBA nume simbolice
serviciu de evenimente	decuplează comunicarea între obiecte CORBA
serviciu de legături	stabileşte legături (cu multiplicități şi tipuri asociate) între obiecte CORBA
serviciu de externalizare	realizează transformarea obiectelor CORBA în / din surse externe
serviciu de tranzacții	coordonează atomicitatea accesului la obiecte CORBA

[5] Acest obiectiv ar presupune că se poate rescrie cu uşurință codul dezvoltat pentru accesarea unui serviciu CORBA al unui producător, astfel încât să poată funcționa 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.

control al accesului concurent	oferă servicii de blocare a accesului la obiecte CORBA pentru asigurarea mecanismului de serializare
serviciu de proprietăți	stabileşte asocierea perechilor de tip nume-valoare cu obiecte CORBA
serviciu de identificare	găseşte 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 un serviciu de rezervare a locurilor la un restaurant, a cărui funcționalitate a fost descrisă pe larg în laboratorul anterior.

3. Specificații IDL

Prin limbajul de definire a 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 realizate[8]. Implementarea unui obiect CORBA este realizată într-un limbaj de programare, interfața fiind doar o punte de legătură (un contract[9]) între codul care foloseşte obiectul[10] şi codul care defineşte obiectul.

Interfața în care este specificat comportamentul obiectului distribuit este independentă de limbajul de programare. IDL defineşte corespondențe pentru mai multe limbaje de programare[11], permițând deci ca implementarea unui obiect să se poată face în limbajul de programare adecvat pentru obiectul respectiv. Totodată, acelaşi lucru este posibil şi pentru client.

Pot fi definite astfel, independent de limbajul de programare, interfețe modularizate pentru obiecte, specificând atribute şi 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 şi 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 şi de ieşire) fără a oferi detalii despre modul de implementare.
[10] Clientul este dependent doar de interfață.
[11] Au fost standardizate corespondențe între IDL şi C, C++, Java, Ada, COBOL, SmallTalk, Objective C, Lisp.

Corespondența între tipuri de date IDL şi 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 şi structuri Java.

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

Un concept întâlnit frecvent atunci când se vorbeşte 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 Reservation
{
    struct Date
    {
        long day;
        long month;
        long year;
    };
    struct Moment
    {
        long hour;
        long minute;
    };
    struct Time
    {
        Date d;
        Moment m;
    };
    struct Interval
    {
        Time start;
        Time end;
    };

    typedef sequence<Interval> Intervals;
```

```
    exception UnspecifiedTimeTable {};

    interface ReservationService {
       Intervals getTimeTable() raises(UnspecifiedTimeTable);
       long getAvailableSeats(in Interval interval)
raises(UnspecifiedTimeTable);
       boolean makeReservation(in long customerId, in Interval interval,
in Interval numberOfSeats) raises(UnspecifiedTimeTable);
       boolean cancelReservation (in long customerId, in Interval interval)
raises(UnspecifiedTimeTable);
    };
};
```

În specificația IDL `Reservation.idl`, au fost definite ca structuri de date `Date`, `Moment`, `Time`, `Interval`, `Intervals`, şi excepția `UnspecifiedTimeTable`, utilizate ulterior în interfața `ReservationService`. Au fost definite mai multe metode ce primesc parametri de intrare (fapt specificat prin cuvântul cheie `in`[12]), descriind funcționalitatea oferită de aplicație.

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

Prin comanda

```
> idlj Reservation.idl
```

vor fi generate următoarele fişiere[14]:

Fişier	Semnificație
`ReservationService.java`	Interfața IDL reprezentată ca intefață Java
`ReservationServiceHelper.java`	Implementează operațiile de tipare pentru interfață (`insert`, `extract`, `read`, `write`, `narrow`, `unchecked_narrow`)
`ReservationServiceHolder.java`	E folosit pentru operații cu parametri de tip `out` şi `inout`.
`ReservationServiceOperations.java`	Conține descrierile operațiilor implementate de interfață (`ReservationService` este derivată din `ReservationServiceOperations`)
`_ReservationServiceStub.java`	Implementează un obiect local reprezentând obiectul CORBA „la distanță” ce transmite cererile spre obiectul distribuit

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

[12] Parametrii de ieşire (indicând date care sunt transmise de la obiectul distribuit către client) sunt specificați prin cuvântul cheie `out`, iar parametrii de intrare ieşire (indicând date transmise de la client către obiectul distribuit şi înapoi la client) sunt specificați prin cuvântul cheie `inout`.
[13] Utilitarul `idlj` se găseşte în directorul `bin` al distribuției standard.
[14] Sunt menționate doar fişierele cu semnificație în dezvoltarea aplicației distribuite, fiind omise fişierele corespunzătoare celorlalte structuri de date, al căror format (ca denumire şi funcționalitate) este asemănător.
[15] Implicit (dacă utilitarul este apelat fără parametri), sunt generate doar fişierele pentru client.

4. Etapele dezvoltării unei aplicații distribuite folosind CORBA

Aşadar, etapele dezvoltării unei aplicații distribuite folosind CORBA sunt:

- definirea interfeței care stabileşte funcționalitatea obiectului distribuit, folosind limbajul de specificare IDL, astfel încât programatorii să poată implementa programe client şi 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 foloseşte `idlj`); se obține versiunea interfeței corespunzătoare limbajului de programare respectiv, şi 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 şi de aşteptare a invocărilor realizate de client;
- dezvoltarea clientului (se folosesc clasele ciot generate); clientul porneşte 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) şi conține serviciile: de inițializare (Bootstrap Service), de nume tranzitoriu[16] (Transient Name Service) dar şi de nume permanent (Persistent Name Service) şi un proces de gestiune pentru server (Server Manager).

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

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

  Într-un sistem Windows, serviciul de nume se porneşte 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** şi **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`. Totuşi, 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`, aşadar 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 cunoaşte doar specificațiile din interfața IDL[17]. 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 moştenirii, care presupune implementarea unei clase care extinde clasa schelet generată de compilator; din acest model fac parte:
  - standardul POA[18] (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 moşteneşte clasa schelet, dar deleagă 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 şi inițializarea instanței ORB;
- obținerea unei referințe către obiectul rădăcină POA şi 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;
- aşteptarea invocărilor obiectului distribuit de către client.

[17] Încapsularea completă a obiectelor distribuite CORBA oferă mai multă libertate în ceea ce priveşte implementarea lor, aceasta putând fi realizată în limbaje de programare diferite chiar şi 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 deserveşte prin intermediul unui obiect referință. POA este un tip particular al acestui tip de obiect definit de specificația CORBA care îndeplineşte 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 – deşi 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 Reservation.*;

public class Server {

    public static void main (String[] args) {

        try {

            ORB orb = ORB.init(args,null);

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

            ReservationServiceImplementation ReservationServiceImpl =
            new ReservationServiceImplementation();
            org.omg.CORBA.Object
            ReservationServiceImplRef =
            POARoot.servant_to_reference(ReservationServiceImpl);
            ReservationService ReservationServiceRef =
            ReservationServiceHelper.narrow(ReservationServiceImplRef);

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

            String serviceName = "ReservationService";
            NameComponent nameComponent =
            new NameComponent(serviceName,"");
            NameComponent path[] = { nameComponent };

            nameContextRef.rebind(path,ReservationServiceRef);

            orb.run();

        } catch (Exception exception) {
            System.out.println ("exception: "+exception.getMessage());
        }
    }
}
```

Clasa `ReservationServiceImplementation` reprezintă servantul, care extinde clasa `ReservationServicePOA`, astfel încât moşteneşte 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 şi fiecare client CORBA. Toate obiectele servant 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 primeşte argumente din linia de comandă, se pot specifica anumite proprietăți, între care cele mai importante sunt portul şi 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 acestuia[19] vor prelucra cererile venite de la client.

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

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) `ReservationServiceHelper`.

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 interfețe[21]. 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 foloseşte 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 şi referința către obiectul distribuit, întoarsă către client prin apeluri tip `NamingContext.resolve_str(serviceName)`, prin care se poate obține acces către funcționalitatea dorită.

Apelul `orb.run()` blochează serverul în aşteptarea 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 aşteptarea unor alte apeluri[22].

[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 găseşte prelucrarea fiecărui obiect POA asociat.

[20] Obiectele pot fi făcute vizibile către client şi prin transformarea într-un şir de caractere a referințelor către acestea şi publicarea lor într-un fişier.

[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 şi inițializarea instanței ORB – se foloseşte metoda `init()` la fel ca în cazul serverului, iar parametrii (din linia de comandă) indică, similar, portul şi 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 şi 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 aceeaşi 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 găseşte o implementare a sa) – se foloseşte metoda `resolve_str` a obiectului care conține o referință contexul de nume, având ca parametru denumirea sub care a fost specificat, în acelaşi 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 şi transmitere fiind transparente către utilizator.

```
import org.omg.CORBA.*;
import org.omg.CosNaming.*;
import Reservation.*;

public class Client {

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

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

            String serviceName = "ReservationService";
            ReservationService ReservationServiceRef =
            ReservationServiceHelper.
            narrow(nameContextRef.resolve_str(serviceName));

            // TO DO: apeleaza metode obiect rezervare
        } catch (Exception exception) {
            System.out.println ("exception: "+exception.getMessage());
        }
    }
}
```

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 şi aplicații comercializate de diverşi 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 şi 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 realizeaza implementarea[23]. Totodată, orientarea pe obiecte este limitată la obiecte transmise prin referință (obiectul nu poate fi transmis între maşini prin cod).

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

Limbajul IDL de specificare a intefețelor este pur declarativ, proiectat pentru specificarea operațiilor corespunzătoare unei aplicații distribuite independent de limbajul de programare. El cuprinde şi corespondențele către limbaje de programare ca Java, C, C++, Lisp, Python, Smalltalk, COBOL şi 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 şi implementarea ei sunt dezvoltate folosind acelaşi limbaj de programare, astfel încât nu există problema corespondenței dintre acestea. Obiecte la nivel de limbaj de programare (codul însuşi) pot fi transmise de la un proces la altul.

Modelul de programare RMI se referă la prelucrări distribuite prin API-ul cu acelaşi 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 şi conține serviciul ORB şi compilatorul `rmic` pentru generarea claselor ciot şi schelet precum şi a legăturilor dintre obiecte la distanță folosind protocoalele JRMP sau IIOP. Folosind acelaşi limbaj de programare, dezvoltarea este mai rapidă, iar flexibilitatea este obținută prin posibilitatea transmiterii de obiecte serializabile.

[23] Tipurile parametrilor de intrare şi de ieşiure trebuie să fie cele specificate în interfață.

Modelul de programare IDL[24] sau RMI cu componenta IIOP vor fi folosite aşadar doar pentru interfațarea cu anumite aplicații (*eng.* legacy systems), preferându-se folosirea modelului de programare RMI din motive ce țin de portabilitate, securitate şi colectarea memoriei disponibile.

Activitate de Laborator

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

În cazul când calea către directorul `bin` al SDK-ului Java nu este precizată în variabila de mediu `PATH`, acest script nu va fi executat cu succes.

Precizarea căii către directorul `bin` al SDK-ului Java în variabila de mediu `PATH` se realizează prin instrucțiunile:


`SET PATH=%PATH%;C:\Program Files\Java\jdk1.7.0_45\bin`


`export PATH=${PATH}:/usr/lib/jvm/default-java/bin`

Verificați că procesul orbd rulează înainte de a porni serverul.

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

```
idlj -fserver Reservation.idl
idlj -fclient Reservation.idl
```

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

Veți observa că la generarea claselor din server lipsesc fişierele `ReservationServiceHelper.java` şi `_ReservationServiceStub.java`. Acestea vor trebui preluate din clasele generate pentru client.

[1p] 3. Să se încarce orarul de funcționare al restaurantului în constructorul clasei `ReservationServiceImplementation` cu datele conținute de fişierul `timetable.txt`.

Se va parsa conținutul fişierului timetable.txt care pe prima linie conține capacitatea (numărul de locuri) încărcată în variabila `numberOfSeats` iar pe următoarele linii programul pentru fiecare zi calendaristică exprimat sub forma `ZZ LL AAAA O1 M1 O2 M2`, acesta fiind încărcat în obiectul `timetable`.

Să se implementeze metoda `getTimetable` a clasei `ReservationServiceImplementation`.

Această metodă întoarce conținutul obiectului `timetable`.

Pentru a reutiliza codul implementat în laboratorul anterior, ar putea fi utilă implementarea unor metode de conversie între clasele Interval / ReservationData deja definite (existente în pachetul entities) şi clasele Interval / ReservationData generate din interfața IDL (existente în pachetul Reservation).

[4p] 4. Să se implementeze metoda `getAvailableSeats` a clasei `ReservationServiceImplementation`.

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

[1p] 6. Să se implementeze metoda `cancelReservation` a clasei `ReservationServiceImplementation`.

[1p] 7. Să se apeleze metodele `makeReservation` şi `cancelReservation` din clasa `Client`. Informațiile utilizate ca parametrii ai metodelor se citesc de la tastatură.

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

[1p] 8. Verificați comportamentul la distanță al aplicației, apelând metodele implementate de colegii voştri.

Pentru ca serverul implementat de colegii voştri să fie vizibil către clientul cu care doriți să îl accesați, componentele trebuie să aibă adrese IP publice sau să se găsească în aceeaşi rețea de calculatoare în cadrul căreia să aibă adrese IP private din acelaşi spațiu (mască de rețea / gateway).

!!! Acest exercițiu nu va fi punctat decât în cadrul laboratorului.

[2p] 9. Să se definească în interfața IDL o metodă

```
Reservations getReservation (in long customerId)
```

care să întoarcă lista cu intervalele în care clientul respectiv are făcute rezervări, precum şi numărul de locuri rezervate în fiecare dintre cazuri.

Să se implementeze metoda în clasa `ReservationServiceImplementations`.

În interfața IDL, o posibilă definiție a structurii Reservations ar putea fi:

```
struct ReservationData
{
    long customerId;
    Interval interval;
    long numberOfSeats;
};

typedef sequence<ReservationData> Reservations;
```

Bibliografie

[1] Introduction to CORBA
http://dsnet.tu-plovdiv.bg/website/container/corba/corba.html
[2] *Orbix, CORBA Programmer's Guide, Java (version 6.3)*, Iona Technologies, 2006