1. CORBA

1.1. Introducere. Scurt istoric. Surse de documentare.

O caracteristică importantă a rețelelor de calculatoare (Internet, Intranet) este heterogenitatea. Aceasta oferă avantajul de a putea alege cele mai potrivite componente software şi hardware pentru diferite părți ale rețelelor şi diferite aplicații, ca şi pe acela de a putea dezvolta rețelele cu echipamentele şi programele cele mai importante la un moment dat. Prin contrast, majoritatea interfețelor de programare a aplicațiilor sunt orientate spre platforme omogene.

Pentru a orienta şi facilita integrarea unor sisteme dezvoltate separat, într-un singur mediu distribuit eterogen, **OMG (Object Management Group)** – consorțiu cuprinzând peste 700 de dezvoltatori, comercianți şi utilizatori de software – a elaborat, adoptat şi promovat standarde pentru aplicații în medii distribuite. Unul dintre ele este **CORBA – Common Object Request Broker Architecture**, care se constituie într-un cadru de dezvoltare a aplicațiilor distribuite în medii eterogene. CORBA este cel mai ambițios proiect al OMG, la care au aderat toate marile firme de software, cu excepția Microsoft, firmă ce a dezvoltat propriul său model, **DCOM (Distributed Component Object Model)**, incompatibil cu CORBA.

CORBA este elementul cheie al OMA – Object Management Architecture, model propus de OMG pentu a transpune în mediu distribuit eterogen toate avantajele programării orientate pe obiecte, incluzând: dezvoltarea rapidă, reutilizabilitatea, protecția implicită, etc.

OMA include Modelul de Obiect (**Object Model**) care precizează cum vor fi descrise obiectele distribuite într-un mediu eterogen şi Modelul de Referință (**Reference Model**) care caracterizează interacțiunile dintre obiecte.

Un **obiect** este o grupare unitară încapsulată de operații şi date, cu identitate distinctă. Un obiect este definit prin **interfața** pe care o prezintă altor obiecte, prin **comportarea** sa la invocarea unei operații şi prin **stare**. Serviciile unui obiect sunt disponibile prin **interfață**, foarte bine definită. Un client, care la rândul său este un obiect, are acces la obiectul server prin invocarea uneia din metodele sale. Relația dintre client şi obiect este caracterizată de aspectele la cere ne referim în continuare.

1.2. Arhitectura sistemelor distribuite în modelul CORBA

Un sistem tipic cuprinde programe **clienți** care utilizează **obiecte** distribuite în sistem. Deoarece în multe sisteme de operare prelucrările trebuie să aibă loc într-un proces, orice obiect trebuie să evolueze într-un proces. În alte cazuri, obiectele pot evolua în thread-uri sau în biblioteci cu legare dinamică (DLLs). CORBA dă factor comun între aceste posibilități şi precizează că obiectele există în **servere** (uneori, în loc de server se mai foloseşte termenul de **implementare**). Fiecare obiect este asociat cu un singur server. Dacă, la invocarea obiectului, serverul nu este în execuție, atunci CORBA va activa serverul, automat.

Codul unui server include implementarea obiectelor asociate, precum şi o funcție principală care inițializează serverul şi crează un set inițial de obiecte. Acestea pot fi de acelaşi tip sau de tipuri diferite. Serverul poate include şi obiecte non-CORBA, de exemplu obiecte C++ accesibile doar din server. Doar obiectele pot fi invocate de clienți, nu şi serverul însuşi.

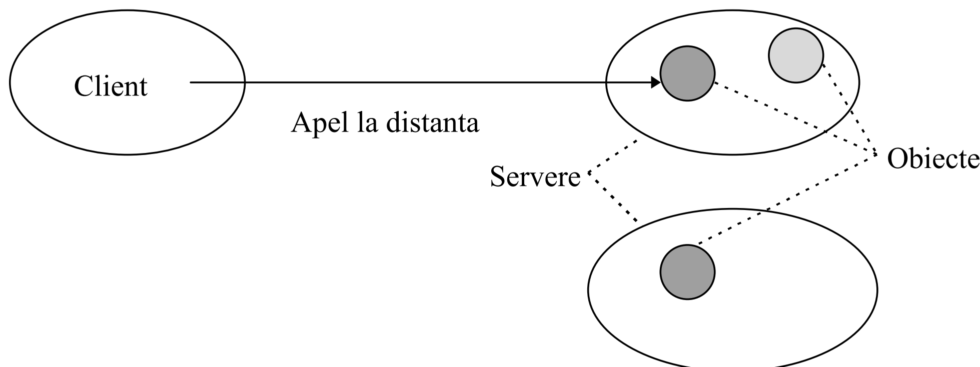

Figura 1.1. Arhitectura CORBA

Obiectele unui server pot invoca obiecte din alte servere. Pe durata unei invocări, serverul respectiv joacă ro de client. Datorită acestei facilităţi, sistemele pot avea arhitecturi foarte diverse, nefiind limitat la arhitectura stea cu un server central la care fac apel mai mulţi clienţi.

Mai multe servere pot coopera, oferind un serviciu global clienţilor. Fiecare server îndeplineşte o funcţie specifică. O altă variantă este cea în care serverele au aceeaşi funcţionalitate. Clientul este pus în contact cu serverul local (aflat în acelaşi sistem de calcul) sau cu serverul cel mai puţin încărcat.

În multe cazuri, codul clientului poate conţine obiecte CORBA, pe care serverul le poate invoca pentru transmiterea informaţiilor despre anumite evenimente sau despre modificarea valorilor unor date. Referinţele acestor obiecte sunt transmise de client serverului ca parametri ai unor apeluri anterioare.

Invocările făcute de clienţi sunt implicit **blocante**: clientul aşteaptă ca cererea să fie transmisă serverului, serverul să execute operaţie invocată şi răspunsul să fie returnat clientului. Sunt posibile şi alte forme:

- într-un apel neblocant, clientul continuă să execute operaţii în paralel cu serverul şi aşteaptă terminarea apelului mai târziu
- un apel **store-and-forward** este mai întâi înregistrat într-o memorie persistentă şi apoi este transmis obiectului ţintă
- într-un apel **publish-and-subscribe** se transmite un mesaj cu un anumit **subiect**, oricare obiect interesat de subiectul respectiv putând primi mesajul.

1.3. Intercomunicarea obiectelor

Implementarea şi localizarea unui obiect sunt ascunse clientului. Comunicarea între client şi obiect este facilitată de **ORB (Object Request Broker)**. Aşa cum sugerează şi numele, ORB permite obiectelor să se regăsească unele pe altele în reţea. El ajută obiectele să facă cereri şi să primească răspunsuri de la alte obiecte aflate în reţea. Acestea se desfăşoară în mod transparent pentru client: el nu trebuie să ştie unde este localizat obiectul, care este mecanismul utilizat pentru a comunica cu el, cum este activat sau memorat acesta, etc.

ORB este mai sofisticat decât formele alternative client/server, incluzând RPC-urile sau comunicarea peer-to-peer. ORB permite ca obiectele să se descopere reciproc la execuţie şi să-şi invoce reciproc serviciile. Din aceste motive, ORB este caracterizat adesea ca fiind o **magistrală de obiecte (object bus)**.

1.3.1. CORBA şi Middleware

Ca o paranteză, serviciile CORBA fac parte dintr-o categorie denumită generic **middleware**. Acesta este un set de servicii independente de aplicaţii care permit aplicaţiilor şi utilizatorilor să interacţioneze prin reţea. În esenţă, middleware este software-ul localizat între reţele şi aplicaţii.

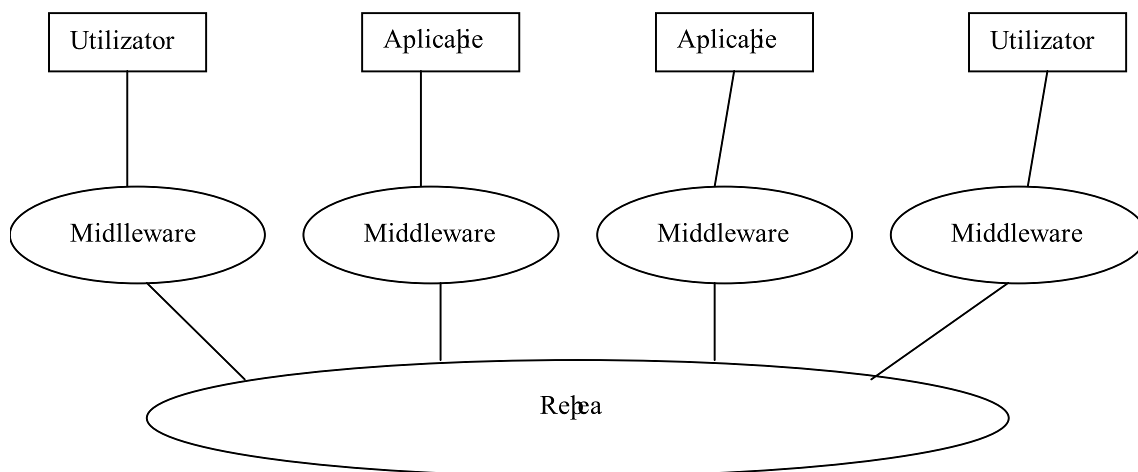

Figura 1.2. Rolul middleware-ului

Serviciile pot fi grupate în trei **categorii**:

- De schimb de informaţie
- Orientate spre aplicaţii (replicarea datelor, integritatea, servicii de grup pentru procese cooperative, servicii pentru obiecte distribuite, etc.)
- De management (directoare, securitate, toleranţă la defecte, performanţă).

Serviciile middleware sunt disponibile aplicaţiilor prin interfeţe de programare a aplicaţiilor, **API** (iar operatorul uman prin **GUI – Graphical User Interfaces**). În practică, serviciile apar ca o combinaţie de componente logice (niveluri), cum sunt cele din figura 1.3.

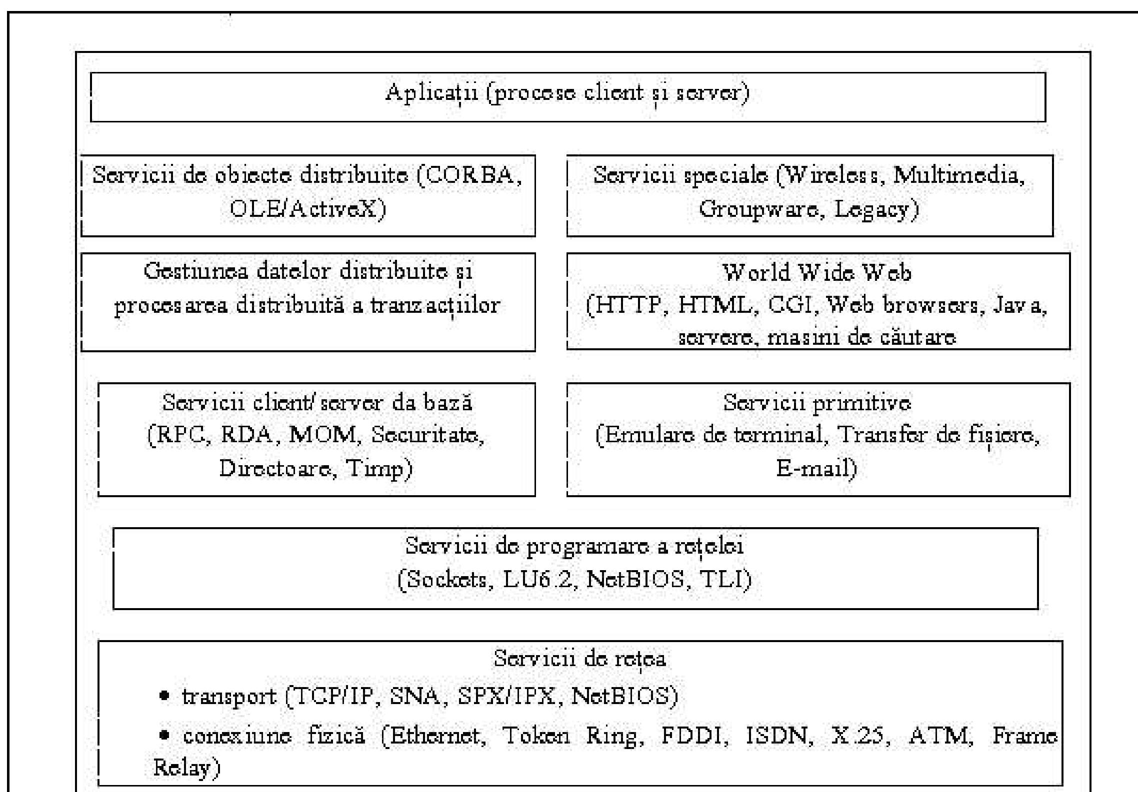

Figura. 1.3. Servicii dezvoltate pe baza modelului Internet

1.3.2. Transparenţa faţă de sistem

Un ORB se poate executa local, pe un singur calculator, sau poate fi conectat cu orice alt ORB din Internet, folosind protocolul **IIOP (Internet Inter-ORB Protocol)** definit în CORBA 2.0. Un ORB

poate transmite apeluri între obiectele unui singur proces, ale unor procese executate pe aceeaşi maşină sau ale unor procese din sisteme diferite, cu sisteme de operare diferite.

1.3.3. Transparența limbajului

Clientul nu ştie cum este implementat obiectul server, în ce limbaj de programare a fost scris. Clientul poate folosi un limbaj de programare la alegere.

CORBA separă interfața de implementare şi foloseşte un limbaj neutru pentru descrierea interfețelor: **IDL – Interface Definition Language**. Acesta este un limbaj pur declarativ ce permite specificarea interfeței şi structurii obiectelor, ce trebuie cunoscute de către clienți. Metodele specificate in IDL pot fi scrise şi invocate în orice limbaj pentru care au fost definite corespondențele cu CORBA: pentru moment C, C++, Ada, Smalltalk, în lucru Cobol, Java Objective C.

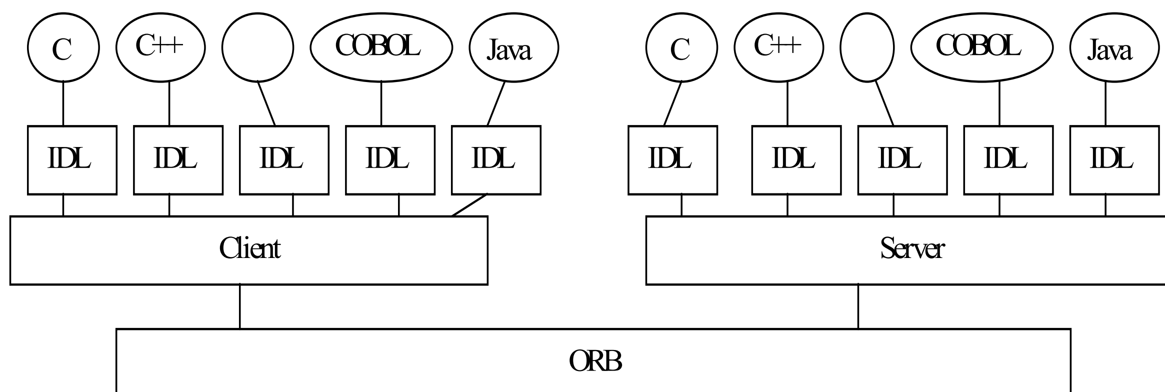

Figura 1.4. Structura unei aplicații client server

Programatorii lucrează cu obiecte CORBA folosind construcții ale limbajului nativ de programare. IDL furnizează interfețe independente de limbajul de programare şi de sistemul de operare, către toate serviciile rezidente în magistrala CORBA. IDL permite interoperarea între clienți şi obiecte server, scrise în limbaje diferite. IDL permite nu doar definirea serviciilor unor obiecte server, dar şi "îmbrăcarea" unor aplicații "moştenite" astfel încât acestea să se comporte ca obiecte. De exemplu, o aplicație scrisă în COBOL se poate comporta ca un obiect server atât timp cât are un IDL şi execută operațiile definite de această interfață.

1.3.4. Invocări statice şi dinamice ale metodelor

ORB permite definirea statică, la compilare, sau dinamică, la execuție a invocărilor de metode. Prima formă beneficiază de toate verificările de tipuri disponibile la compilare; a doua are o mai mare flexibilitate.

1.3.5. Sisteme autodescriptibile

CORBA furnizează metadescrieri ale serviciilor disponibile la execuție. Acestea sunt păstrate într-un depozit de interfețe, **Interface Repository**. Clienții găsesc în aceste depozite informații despre modul în care trebuie invocate serviciile, la execuție.

1.3.6. Exemple de ORB

- **DSOM** - Distributed System Object Model (IBM) şi Neo (Sun) sunt ambele compatibile cu CORBA. Ele folosesc IIOP pentru comunicarea între obiecte.
- **DCOM** – Distributed Common Object Model (Microsoft) este un ORB dar nu este compatibil cu CORBA.

1.4. Modelul de referință

1.4.1. Servicii

CORBA depăşeşte nivelul facilităţilor de inter-operare a obiectelor. El specifică, în plus, un set cuprinzător de **servicii** pentru crearea şi desfiinţarea obiectelor, accesul lor prin nume sau proprietăţi, depunerea lor în memorii persistente, definirea de relaţii între ele, etc.

Prin aceasta, CORBA permite crearea unui obiect obişnuit, cu o funcţionalitate legată de o anumită aplicaţie, şi adăugarea ulterioară (prin moştenire multiplă de la serviciile corespunzătoare) a securităţii, protecţiei, persistenţei, tranzacţionalităţii, etc.

Categoriile de interfețe de servicii sunt prezentate in figura 7.3.

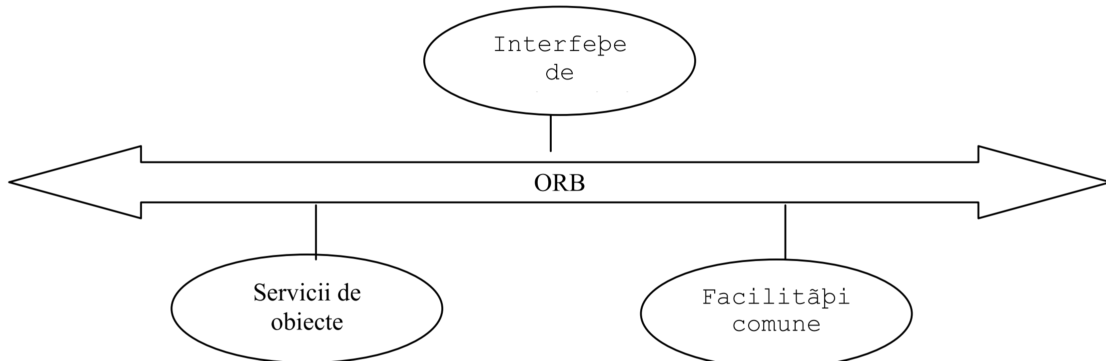

Figura 1.5. Categorii de interfețe

Serviciile de obiecte completează funcţionalitatea ORB. Au fost publicate standarde pentru 16 servicii de obiecte grupate în următoarele categorii:

- sisteme distribuite (D)
- baze de date (B)
- generale (G).

Dintre cele mai importante, amintim următoarele:

- (D) Serviciul de nume (Naming Service) permite obiectelor să găsească alte obiecte, pe baza numelor;
- (D) Serviciul de găsire a obiectelor după proprietăţi (Trading Service)
- (G) Serviciul ciclului de viaţă (Life Cycle Service) defineşte operaţiile de creare, copiere, mutare şi ştergere a componentelor pe magistrala de obiecte.
- (B) Serviciul de persistenţă (Persistence Service) oferă o singură interfaţă pentru memorarea persistentă a obiectelor într-o varietate de servere de memorare, incluzând baze de date de obiecte, baze de date relaţionale sau simple fişiere.
- (D) Serviciul de evenimente (Event Service) permite obiectelor să înregistreze sau să anuleze interesul lor pentru evenimente specifice. Serviciul defineşte un obiect denumit **event channel** care colectează şi distribuie evenimente pentru obiecte care nu se cunosc reciproc.
- (B) Serviciul de proprietăţi (Properties Service) permite asocierea unor perechi (nume-valoare) cu obiectele.
- (G) Serviciul de timp (Time Service) permite sincronizarea în funcţii de timp şi gestiunea evenimentelor dependente de timp.
- (D) Serviciul de securitate (Security Service) oferă un cadru complet pentru securitatea obiectelor distribuite. Suportă **autentificarea**, **liste de control al accesului**, **confidenţialitatea** şi **ne-repudierea**. De asemenea, **delegarea de acreditive** între obiecte.

- (B) Serviciul de control al concurenței (Concurrency Control Srrvice) permite gestiunea blocărilor/deblocărilor.

Transaction Service, Relationship Service, Querry Service reprezintă alte servicii importante de baze de date.

1.4.2. Facilitățile comune

Furnizează servicii orientate spre aplicații. Un exemplu este DDCF – Distributed Document Component Facility, bazat pe OpenDoc (Apple Computer). Acesta permite prezentarea şi interschimbul de obiecte bazate pe un model de document. El facilitează, de exemplu, legarea unui obiect foaie de calcul într-un raport.

Facilitățile comune aflate în construcție include: agenți mobili, interschimb de date, cadre de obiecte comerciale, internaționalizarea.

1.4.3. Interfețele de aplicații

Se referă la obiecte specifice unei anumite aplicații. Ele nu sunt standardizate şi nu prezintă interes pentru OMG decât în măsura în care anumite servicii depăşesc (treptat) domeniul unei singure aplicații. În acest sens, **obiectele comerciale** – business objects – reprezintă o cale naturală de descriere a conceptelor independente de aplicație, cum ar fi client, ordin, bani, plată, pacient, automobil, etc. Noțiunea de obiect comercial trebuie acceptată într-un sens larg, ea referindu-se la o componentă ce reprezintă o entitate "recognoscibilă" în lumea reală. O colecție va consta dintr-o colecție de obiecte comerciale ale căror interacțiuni şi comportări imită entitățile lumii reale pe care o modelează.

1.5. Arhitectura CORBA ORB

CORBA ORB este partea ce mijloceşte stabilirea relațiilor client/server între obiecte. În 1995, OMG a publicat CORBA 2.0, ale cărui principale caracteristici sunt evidențiate în continuare. Structura sa este prezentată in figura 8.1.

Folosind ORB, un obiect client poate invoca o metodă a unui obiect server, în mod transparent. ORB interceptează apelul şi este răspunzător de găsirea obiectului server, de transmiterea parametrilor şi de invocarea metodei, ca şi de returnarea rezultatelor.

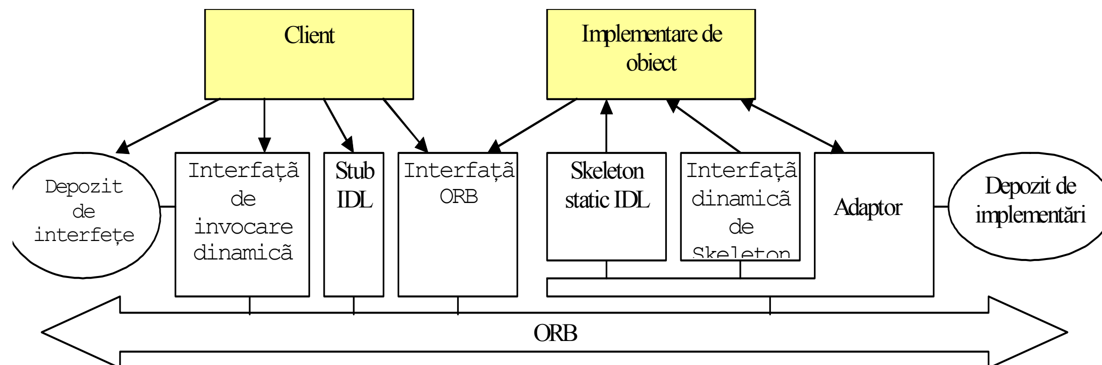

Figura 1.6. Arhitectura CORBA

Clientul nu trebuie să cunoască localizarea obiectului server, limbajul de programare în care a fost scris, sistemul de operare în care se execută sau orice alte aspecte care nu sunt cuprinse în interfața obiectului. Este important de notat că rolurile de client şi de server sunt determinate de modul de coordonare a interacțiunilor între obiecte şi se pot modifica în timp.

1.5.1. Nucleul ORB

Obiectul căruia ORB trebuie să-i transmită cererea clientului se numeşte **obiect-țintă**. Caracteristica ORB este transparența cu care asigură comunicarea clientului cu ținta. El **ascunde**:

- Localizarea obiectului – în acelaşi proces, în procese diferite ale aceleaşi maşini, în noduri de rețea diferite

- Implementarea obiectului – (limbaj, SO, calculator)
- Starea de execuție: clientul nu trebuie să ştie dacă obiectul server este **activat** şi gata să primească cererea; dacă este cazul, ORB porneşte obiectul înainte de a-i da cererea.
- Mecanismul de comunicare cu obiectul – TCP/IP, cu memorie partajată, apelul unor metode locale, etc.

Clientul specifică obiectul țintă printr-o **referință de obiect**. Aceasta se creează odată cu obiectul şi se referă la el în mod unic, atâta timp cât obiectul există. Referința este "**opacă**" pentru client, în sensul că acesta nu o poate modifica. Referințele de obiecte sunt gestionate exclusiv de ORB şi au formate standard (ca în IIOP) sau proprietare (de firmă).

Clienții pot obține referințele de obiecte pe mai multe căi:

- La crearea obiectului – clientul poate crea un obiect şi obține referința sa. CORBA nu are o operație specială de creare de obiecte. Pentru a crea un obiect, clientul invocă o operație a unui obiect **fabrică de obiecte**. Crearea întoarce o referință de obiect.
- Prin **serviciul de cataloage** (directory service) – ambele Naming Service si Trading Service permit obținerea de referințe, după nume, respectiv după proprietăți.
- Prin conversia la/de la şiruri de caractere – o referință de obiect poate fi convertită în şir de caractere şi înapoi în referință putând fi utilizată după conversie, atât timp cât obiectul există.

Problema: cum se lansează aplicația şi cum poate obține o referință inițială de obiect. **Soluția**: ORB are un serviciu simplu de nume, incorporat, care furnizează aplicației referințele unor servicii mai generale (Naming – Trading). Operația **resolve-initial-reference** cu parametrul **Name-Service** furnizează aplicației referința de obiect pentru serviciul de nume cunoscut de ORB.

2. OMG IDL - Interface Definition Language

Pentru ca un obiect să adreseze cereri unui obiect (server), el trebuie să cunoască tipurile **operațiilor** suportate de obiect. Acestea sunt precizate de specificația interferenței obiectului. Pentru a realiza independența față de limbajul de programare în care este descris obiectul, interfața este definită într-un limbaj neutru, IDL – Interface Definition Language. Interfețele sunt similare claselor în C++ şi interfețele în Java.

IDL prevede mai multe declarații pentru: tipuri de bază (short, long, char, etc), tipuri template (string, sequence), tipuri construite (struct, union, etc)

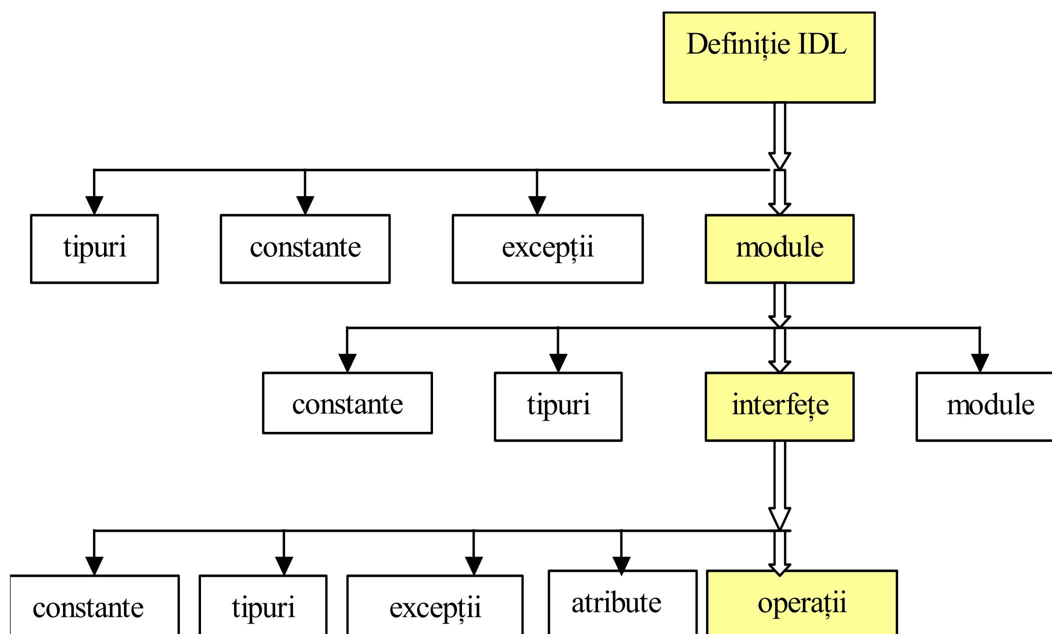

Figura 2.1. Declarații IDL

Acestea sunt folosite în declarațiile **operațiilor** pentru a defini tipurile argumentelor şi rezultatului. La rândul lor, operațiile sunt folosite în declarațiile de **interfețe**, pentru a defini serviciile furnizate de obiecte. Un ansamblu de interfețe şi definiții de tipuri poate constitui un **modul**, al cărui rol este legat de domeniile de valabilitate ale numelor declarate. Un fişier IDL formează un domeniu de valabilitate al numelor (scope). Un nume poate fi definit o singură dată într-un domeniu. Oricum, numele poate fi re-definit într-un subdomeniu inclus, cum ar fi cel corespunzător unui modul. Forma generală a declarației unui modul este următoarea:

```
module modulename
{       interface interfacename1 {…};
        interface interfacename2 {…};
}
```

O specificație constă din zero sau mai multe definiții de tip, constante, excepții sau module. Un modul poate conține, în afara interfețelor şi definiții de tipuri şi alte module.

2.1. Tipuri

2.1.1. Tipuri de bază

IDL suportă următoarele tipuri de bază:

- long (signed, unsigned) – 32 biți
- long long (signed, unsigned) – 64 biți
- short (signed, unsigned) – 16 biți
- float, double, long double
- char, wchar (wide char – caractere de 16 biți)
- boolean
- octet (8 biți) – se garantează că nu sunt aplicate conversii la transmiterea prin rețea cu ORB.
- enum
- any – poate reprezenta orice tip de bază sau construit de utilizator.

2.1.2. Tipuri construite

Exemplu structură.

```
        struct catalog {        course curs;
                                grade nota;
}
```

Exemplu union cu discriminant:

```
union DataItem switch (char){   case ‘c’: long count;
                                case ‘a’: float amount;
                                default: char initial;
}
```

Exemplu tablou de lungime fixă, cu elemente de un singur tip:

```
typedef char message[80];
```

2.1.3. Tipuri template

- string şi wstring mărginite şi submărginite

```
string                          // un sir fără limitări de lungime
string<10>                      // un sir de maximum 10 caractere
```

- sequence tablou unidimensional, mărginit sau nemărginit, cu toţi membrii de acelaşi tip

```
sequence<float, 100> MySeq;         //mărginit
sequence <float> Seq;               //nemărginit
```

2.2. Constante

Se pot defini constante simbolice, ca în orice limbaj de programare, oriunde într-un fişier IDL. IDL calculează valoarea constantelor folosind:

Operatorii sunt

- unari -, +, ~ (bit complementat)
- binari *, /. %, +, -, <<, >> (shift), & (şi pe biţI), | (sau pe biţI), ^ (xor pe biţI)

De exemplu:

```
const long double_size = max_size*2;
const string<5> help_message="help";
```

2.3. Definiţii de interfeţe. Moştenirea.

O interfaţă este o descriere a operaţiilor pe care un obiect le poate executa. Un obiect **satisface** interfaţa dacă el poate fi specificat ca ţintă în oricare cerere potenţială descrisă de interfaţă. Forma generală este:

```
interface nume[:baza {, baza}]{
..: declaraţii de constante, tipuri, excepţii, atribute, operaţii, …
};
```

Fiecare interfaţă defineşte un nou **tip de referinţă de obiect** şi un nou domeniu de valabilitate a numelor (scope). O definiţie de interfaţă poate conţine declaraţii de: **constante**, **tipuri**, **excepţii**, **atribute** şi **operaţii**.

```
Exemple: interface Factory{
                    Object create();
    };
```

defineşte o interfaţă numita **Factory**, care are o operaţie **create**. Operaţia nu are parametrii şi întoarce o referinţă de obiect de tip **Object**. Dată fiind o referinţă de obiect de tip Factory, un client poate invoca operaţia pentru a crea un nou obiect CORBA.

O interfaţă poate moşteni de la una sau mai multe alte interfeţe. Moştenirea permite reutilizarea interfeţelor existente pentru a defini servicii noi.

Exemplu

```
        interface Person {
                Readonly attribute string name;
        };
        interface student: Person{
                attribute Profesor advisor;
                Enrollments load (in semester when);
        };
```

student are atributele **name** şi **advisor** şi operaţia **load**.

Exemplu

```
interface SpredSheetFactory: Factory {
        Spreadsheet create_spreadsheet();
};
```

Aici, SpreadSheet are două operații, **create** moştenită de la Factory şi **create_spreadsheet** definită direct în interfață.

Moştenirea este un concept important în CORBA, care se supune **principiului Open-Close**: el permite ca sistemul să fie **deschis** extensiilor şi **închis** modificărilor. O interfață poate moşteni de la mai multe alte interfețe, dar **nu poate redefini atributele moştenite sau operațiile moştenite**. În schimb o interfață derivată poate redefini orice constantă, excepție, tip care a fost moştenită.

Folosirea numelui interfeței şi a operatorului de rezoluție poate rezolva **ambiguitățile**. Exemplu:

```
interface Foo {typedef short myType;};
interface Bar: Foo {
        typedef long myType;
        void my_op(in myType a, in Foo::myType b);
};
```

În acest exemplu, se copiază tipul **myType** definit în interfață pentru primul parametru şi tipul **myType** definit in Foo pentru al doilea parametru.

Operatorul de rezoluție poate fi folosit şi în cazul în care o constantă, tip sau excepție este definită în mai multe interfețe specificate ca bază pentru interfața curentă. Oricum, **nu se poate moşteni** de la două interfețe care au definit o aceeaşi operație sau un acelaşi atribut.

OMG IDL are un caz special de moştenire: toate interfețele sunr derivate din interfața Object definită în modulul CORBA. Deoarece această moştenire din CORBA::Object este automată, ea nu trebuie specificată explicit pentru fiecare interfață OMG IDL.

2.4. Operații IDL. Operații oneway

Operațiile sunt similare declarațiilor de funcții C/C++. O operație denotă un **serviciu** care poate fi cerut obiectului şi este descrisă prin:

- identificatorul operației
- tipul rezultatului (orice tip IDL)
- descrierea fiecărui parametru
- nume
- tip
- (mod) direcție
    - **in** client->server
    - **out** server->client
    - **inout** ambele direcții

Excepțiile pe care operația le poate provoca (clauza **raises**), ele constituie indicații că operația care le-a generat nu a fost executată corect. Sintaxa unei operații este următoarea:

```
result_type op_name (
        [direction type name {, directions type name} ])
   [raises ([exception {, exception }])]
```

unde **exception** este o excepție definită anterior. De exemplu:

```
Students roster (in Semester when);
void enroll (in Course course_to_take, out Course prereqs);
```

```
        long pop() raises (underflow);
```

Oneway este o caracteristică suplimentară a unei operații. Ea precizează că utilizatorul nu se blochează în operație şi că sistemul nu garantează că cererea este transmisă obiectului țintă. O operație **oneway**: (1) nu poate conține parametrii **out** sau **inout**; (2) trebuie să nu întoarcă un rezultat (tipul rezultatului, void); şi (3) să nu conțină clauza **raises**.

2.5. Excepții şi atribute

În ce priveşte **excepțiile**, ele arată producerea unor erori la execuția unei operații. Excepțiile pot fi clasificate în:

- excepții definite de sistem
- excepții definite de utilizator

O **excepție de sistem,** este generată când se produce o eroare în infrastructura ORB sau la producerea unei noi erori generice în timp ce serverul încearcă să prelucreze o cerere. De xemplu, "Out of memory" sau "Illegal parameter".

O excepție de sistem constă dintr-un **motiv major** (CORBA:: BAD_PARAM) şi un **cod minor**. Uzual, o funcție de conversie **exception_to_string** permite conversia motivului şi a codului într-o formă descriptivă, texturală.

O **excepție utilizator** este definită la fel ca o înregistrare. De exemplu,

```
                exception UnknownId {}:
                exception NotInterested {string explanation};
```

Un alt exemplu:

```
interface FrontOffice{
        ...
        exception NoSuchPlace{
                        Place whereş
        };
        Exception InvalidDate{
                        Date when;
        };
        Places getPrice (in Place chosenPlace, in Date when)
                raises (NoSuchPlace, InvalidDate);
};
```

Atributele interfeței. Pe lângă operații, o interfață poate avea atribute. Un atribut caracterizează **starea** unui obiect, într-o manieră abstractă. El este logic echivalent cu declararea unei perechi **get/set** de funcții de acces la valoarea atributului. Un atribut poate fi **read-only**, caz în care este accesibil doar prin **get**. Forma generală este [readonly] attribute <datatype><name>. De exemplu,

```
        interface Adder {
                        …
                        readonly attribute long sum;
};
```

2.6. Particularități IDL

OMG IDL are un sistem de tipuri simplu, pentru a facilita corespondența cu multe limbaje de programare. Cu toate acestea, sistemul de tipuri este suficient pentru cele mai multe aplicații distribuite. Simplitatea este critică pentru utilizarea CORBA ca o tehnologie integratoare. Dintre elementele pe care le găsim în limbajele de programare şi **nu le regăsim in IDL**, amintim:

- toate definițiile dintr-o interfață IDL sunt publice (nu există conceptul de **private** sau **protected**, acestea fiind legate de implementare nu de specificare)
- nu pot fi declarate variabile-membre; atributele sunt diferite de variabile, ele reprezentând **cererile** pe care un client le poate adresa obiectului şi nu memoria pe care obiectul trebuie să o aibă; un atribut poate fi implementat ca o variabilă, dar şi ca o funcție de alte variabile din sistem
- nu există constructori şi destructori
- nu există supraîncărcarea bazată pe **semnătura** operațiilor
  - semnătura constă din
    - ♦ specificarea parametrilor şi a rezultatului
    - ♦ specificarea excepțiilor şi a tipului datelor care se asociază excepțiilor
    - ♦ specificarea informației suplimentare de context, care poate afecta cererile
  - ♦ specificarea semanticii operației, aşa cum apare ea clientului
- nu există tipuri parametrice
- nu există ierarhii de excepții
- aspecte semantice nu sunt incluse în interfețe
  - ♦ domenii de valori pentru date
  - ♦ ordonarea legală a operațiilor
  - ♦ constrângeri de timp/spațiu asupra implementărilor
  - ♦ garanții tranzacționale ale interfeței

În particular, menționăm diferențele față de C++:

- nu există tipurile int, unsigned int
- char nu poate fi signed sau unsigned
- parametrii
  - trebuie sa aibă nume
  - trebuie să aibă direcție
  - nu pot fi opționali
  - nu pot avea valori explicite
- specificarea tipului rezultatului este obligatorie
- nu există structuri şi uniuni anonime
- nu există pointeri
- nu se poate face supraîncărcarea operatorilor

2.7. Corespondența între IDL şi C++

Corespondența între IDL şi C++ este definită de standardul CORBA. Ea se referă la modul în care definițiile IDL sunt traduse în C++ şi la regulile pe care clienții şi serverele C++ trebuie să le respecte la utilizarea, respectiv implementarea interfețelor.

2.7.1. Tipuri de bază

Corespondențele sunt prezentate în tabelul 2.1.

IDL	C++ typedefs	corespondența în Orbix pentru 32 de biți
short	CORBA::Short	short, dacă 16 biți
long	CORBA::Long	long, dacă 32 de biți
unsigned short	CORBA::UShort	unsigned short
unsigned long	CORBA::ULong	unsigned long
float	CORBA::Float	float
double	CORBA::Double	double
char	CORBA::Char	char, dacă 8 biți
boolean	CORBA::Boolean	unsigned char
octet	CORBA::Octet	unsigned char, dacă 8 biți

Tabel 2.1. Corespondența IDL - C++ pentru tipurile de bază

Observații.

- Între IDL şi tipurile C++ se fooseşte un nivel intermediar, deoarece limbajul C++ nu defineşte detalii despre cerințele de memorie ale multor tipuri (de exemplu, faptul că **short** ocupă 16 biți), în timp ce IDL cere aceste detalii pentru ca aplicațiile să poată inter-opera pe maşini diferite. Corespondența între IDL şi C++ foloseşte deci **typedefs**, cum e **CORBA::Short**, urmând ca implementarea ORB pe fiecare platformă să asigure corespondența corectă.
- Definițiile de tipuri sunt puse într-o **clasă** C++ sau într-un **namespace** (cu numele CORBA).
- Pentru **boolean** s-a ales în Orbix **unsigned char** şi nu **bool** din ANSI C++ pentru că ANSI nu e o cerință pentru CORBA în acest moment.

Regula de **transmiterea parametrilor** prntru tipurile de bază este simplă:

- parametrii **in** şi rezultatele se transmit prin **valoare**
- parametrii **out** şi **inout** se transmit prin **referință** (&)

2.7.2. Module

Modulele IDL sunt mapate pe **namespace** în C++ (o tratare specială este necesară când compilatorul nu suportă namespace). De exemplu:

```
module M{
typedef float money;
        interface I{...
        }ş
};

se traduce prin:

namespace M{
        typedef CORBA::Float money;
        class I: public virtual CORBA::Object{...
        };
};
```

2.7.3. Interfețe

O **interfață** IDL este pusă în corespondență cu o clasă C++. Fiecare **operație** este mapată pe o funcție-membră. Un **atribut** este mapat pe două funcții (de citire şi de modificare a valorii). Un atribut **readonly** este mapat pe o funcție de citire a valorii. De exemplu:

```
interface T{
	attribute long l;
	readonly attribute char c;
	void op1();
};
```

se mapează în C++ pe

```
class T: public virtual CORBA::Object{
  CORBA::Long l()			//get
	throw (CORBA::SystemException);
void l (CORBA::Long)		//modify
	throw (CORBA::SystemException);
CORBA::Char c()			//get
	throw (CORBA::SystemException);
virtual void op1()
	throw (CORBA::SystemException);
};
```

Clasa T moşteneşte din CORBA::Object care este clasa rădăcină pentru toate obiectele CORBA. Tipul (referinţă) CORBA::Object—ptr sau CORBA::Object—var corespunde referinţei oricărui obiect CORBA.

2.7.4. Referințe la obiecte

În C++, referinţele la obiecte de un tip **obT** au tipul **obT_ptr** sau **obT_var**. O variabilă de tip **obT_ptr** se compoirtă ca un pointer normal, deci referinţele la obiect pot fi invocate cu operatorul ->. Utilizarea acestui tip (_ptr) reclamă gestiunea **explicită** a contorului de referinţe asociat fiecărui obiect CORBA. Contorul de referinţe se păstrează local, **separat** pentru obiectul ţintă (din server) şi separat pentru reprezentantul său (proxy) din client.

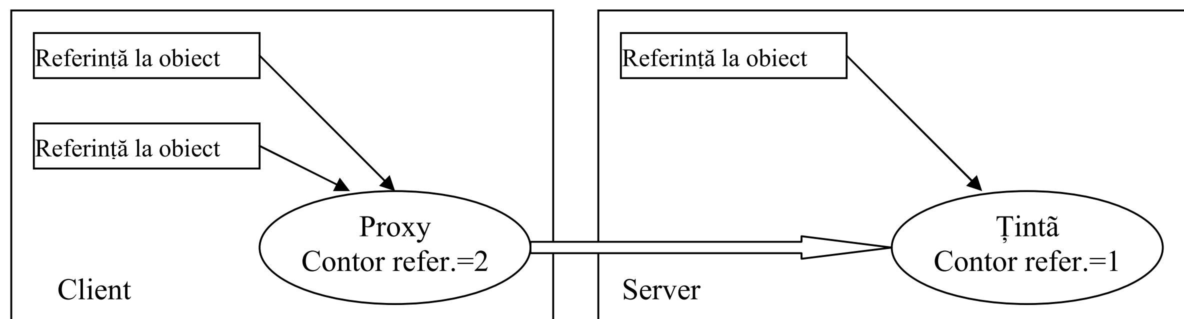

Figura 2.2. Referințe la obiecte

Contorul de referinţe trebuie incrementat la crearea unei noi referinţe la obiect şi decrementat la ştergerea referinţei:

```
obT_ptr p1=...;
{
	obT_ptr p2;
	p2 = obT::_duplicate(p1);			//incrementeaza contorul
	... //poate folosi p2
	CORBA::release (p2);				//decrementeaza contorul
```

```
}
// foloseste apoi p1
CORBA::release(p1)
```

Gestiunea contorului de referințe este automată în cazul tipului _var, considerat ca un pointer "inteligent". De exemplu:

```
obT_var p1=...;
{
        obT_var p2;
        p2=p1;                          //incrementează automat contorul
        ... //poate folosi p2
}
...// fooseste p1, apoi decrementeaza automat contorul cand p1 iese //din domeniul de valabilitate
```

3. CORBA static

CORBA ORB suportă două tipuri de apeluri client/server: statice şi dinamice. În ambele cazuri, clientul execută o cerere atunci când are acces la o referință a obiectului şi specifică metoda care corespunde serviciului. Serverul nu poate face diferențierea între invocările statice şi dinamice. Ne referim aici la prima formă.

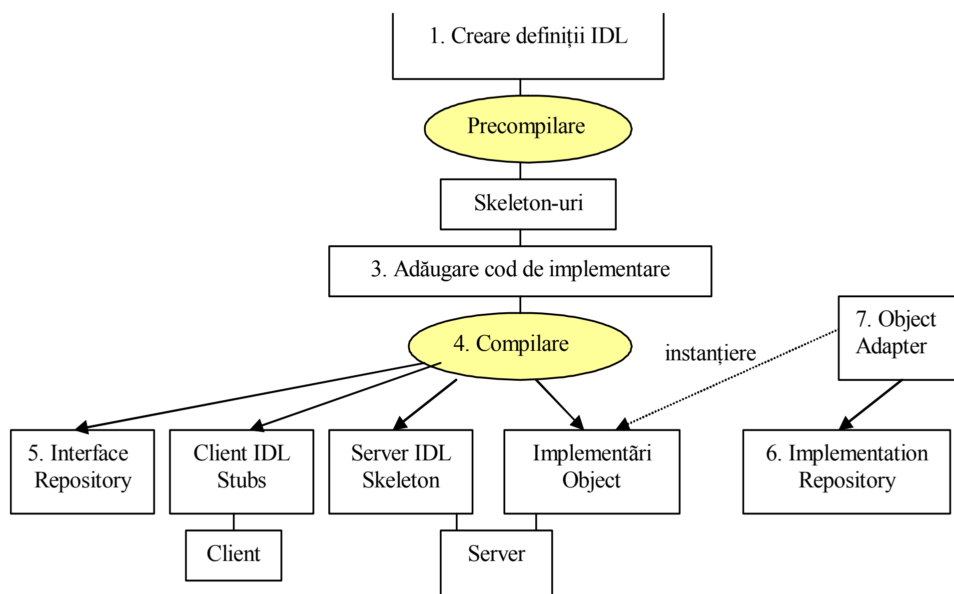

Figura 3.1. CORBA Static

Clienții "văd" interfețele obiectelor prin perspectiva corespondenței între limbajul de implementare şi IDL. **Interfețele statice** sunt generate direct, în forma unor **stub**-uri client, de precompilatoare IDL. Ele au unele avantaje:

- programare mai uşoară
- verificări mai robuste de tipuri
- execuție simplă
- autodocumentare

În schimb nu sunt la fel de **flexibile** ca apelurile dinamice.

3.1. Etapele de dezvoltare a aplicației

Figura 3.1 arată paşii necesari pentru crearea unui server şi a unui client care comunică prin ORB. Figura se referă la cazul general, nu neapărat la varianta statică.

1. Se definesc interfețe în IDL; acestea precizează ce operații sunt disponibile la un server şi cum pot fi invocate.
2. Pe baza descrierii IDL, un precompilator produce **skeleton**-uri (pentru server) şi **stub**-uri (pentru clienți). Când clientul şi obiectul țintă sunt în acelaşi spațiu de adrese, comunicarea lor se poate face direct, nefiind necesar un cod suplimentar. Când aceştia sunt în spații diferite, este necesar un cod suplimentar la client (stub) pentru transmiterea invocărilor, şi la obiectul țintă (skeleton), pentru recepția lor şi transmiterea lor către obiectul țintă.
3. Se adaugă codul care implementează serverul.
4. Se face compilarea codului. Un compilator care acceptă CORBA este, în mod obişnuit, capabil să genereze cel puțin trei fişiere:
   a. Fişiere import – care descriu obiectele pentru Interface Repository
   b. Stub-uri client – pentru metodele definite în IDL; ele sunt invocate de clienți pentru acces la server
   c. Skeleton-uri server – care apelează metodele serverului (mai sunt numite up-call interfaces).
5. Leagă definițiile de interfețe de InterfaceRepository (se foloseşte un utilizator). Informația din IR este accesibilă clienților la execuție.
6. Adaptorul de obiecte înregistrează în Implementation Repository tipul şi referința oricărui obiect ce poate fi instanțiat pe server. ORB foloseşte aceste informații pentru a localiza obiectele active sau să ceară activarea unor obiecte.
7. Instanțierea obiectelor pe server – cerută de object adapter conform unei anumite strategii.

La aceste etape se adaugă operațiile legate de client, la care ne referim în exemplul care urmează.

Paşii de programare ce corespund dezvoltării unei aplicații client-server în CORBA şi C++, în varianta statică, sunt următorii:

- descrierea interfețelor în IDL
- implementarea interefețelor în C++
- scrierea funcției principale a serverului, care crează instanțe ale claselor, informează apoi broker-ul şi adaptorul că au fost făcute inițializările şi că obiectele țintă sunt gata să primească cereri
- scrierea clientului care se conectează la server şi foloseşte serviciile acestuia.

3.2. Specificarea unei aplicații elementare

Descrierea care urmează se referă la **VisiBroker for C++** (produs de Visigenic), dar ea corespunde, cu mici modifcări şi altor implementări CORBA 2.0 (vezi Mico de la Universitatea Frankfurt, Germania).

Programul **Count** folosit aici ca exemplu este o aplicație rudimentară client/server. **Serverul** suportă o singură metodă numită **increment**, care incrementează valoarea unei variabile numită **sum** şi întoarce clientului valoarea acesteia.

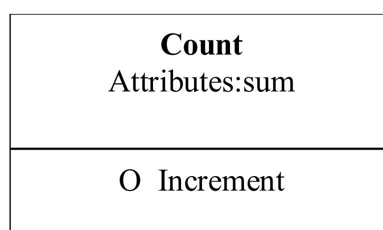

Figura 3.2. Serverul

Sum este declarată ca un atribut read/write. Ca urmare, valoarea sa este accesibilă prin funcții predefinite. Clienții folosesc aceste funcții pentru a stabili valoarea inițială pentru **sum** şi pentru a găsi valoarea finală.

Clientul trebuie să facă următoarele operații:

- Să stabilească valoarea inițială a atributului **sum**
- Să invoce metoda **increment** de 1000 de ori
- Să afişeze valoarea finală a atributului **sum**, împreună cu timpul de răspuns mediu.

Clientul trebuie să poată trata excepțiile de sistem CORBA.

Primul pas este definirea în IDL a **interfeței serverului**. Fişierul **count.idl** conține această descriere:

```
module Counter
{ interface Count
   { attribute long sum;
     long increment();
   };
};
```

3.3. Compilarea interfeței

În plus față de **generarea tipurilor în limbajul de programare dorit**, un compilator IDL generează **stub**-uri client şi **skelton**-uri server. **Stub**-ul este un mecanism care creează efectiv şi generează cereri în numele clientului. **Skeleton**-ul este un mecanism care livrează cererile către implementarea obiectului. Deoarece sunt obținute direct din interfața IDL, ambele sunt (în mod natural) **specifice interfeței**.

O **invocare** de operație asupra unui obiect țintă se prezintă în C++ în forma apelului **unei funcții membre a unei clase**. Acest apel invocă un stub. Deoarece stub-ul este un "reprezentant" local al unui obiect țintă (posibil) distant, el se mai numeşte **intermediar** (**proxy**) sau surogat. **Stub**-ul lucrează direct cu ORB pentru a aranja (marshal) cererea, adică pentru a converti de la forma proprie limbajului de programare la una potrivită pentru transmitere prin magistrala de obiecte, către țintă.

Odată ajunsă la țintă, ORB şi skeleton-ul cooperează pentru a re-aranja (unmarshal) cererea, deci pentru a o converti de la forma transmisibilă la forma unui limbaj de programare. Îndată ce obiectul termină cererea, răspunsul este transmis pe calea inversă: prin skeleton, ORB-ul serverului, conexiune, ORB-ul clientului, stub, aplicație client.

Compilatorul VisiBroker pentru C++ (numit **Orbeline**) produce patru fişiere, pe baza descrierii precedente (vezi Figura 3.3).

Observație. MICO produce două fişiere, care reunesc conținutul celor patru menționate aici.

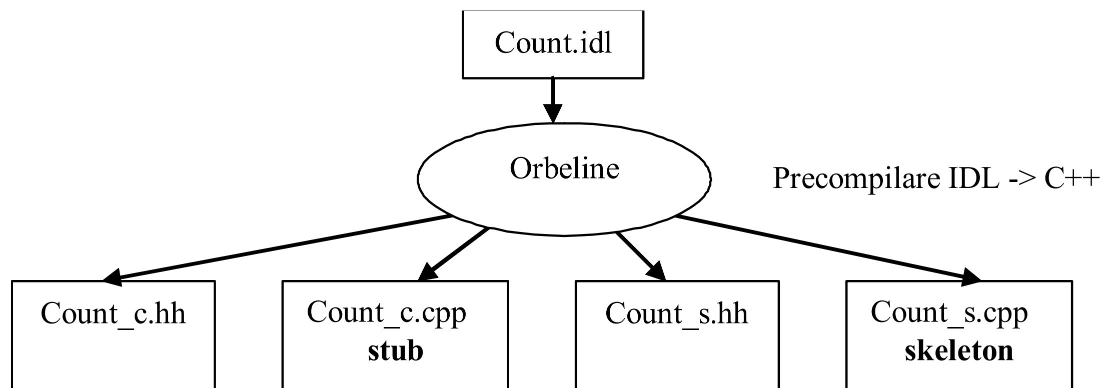

Figura 3.3. Fişiere produse de compilatorul Orbeline

- ♦ **count_s.cpp** – este **skeleton**-ul server pentru metodele **clasei count**. Acesta reprezintă codul care se re-aranjează (unmarshals) apelurile pentru obiectul **Count** şi **invocă implementarea obiectului**.

- **count_s.hh** este fişierul antet pentru server, care include definițiile de clase pentru skeleton-ul implementat în count_s.cpp.
- **count_c.cpp** conține o clasă numita Count care serveşte ca intermediar (proxy) al clientului pentru obiectul **Count**. El conține funcții **stub** şi de aranjare (marshaling) pentru toate metodele definite în interfața **Count**. În plus, implementează o **metodă bind** care ajută clientul să localizeze serverul **Count**.
- **count_c.hh** este fişierul antet pentru client, care include declarațiile şi definițiile de clase pentru stub-ul implementat în count_c.cpp.

Cele patru fişiere generate de compilatorul IDL conțin în cea mai mare parte funcții private ale VisiBroker-ului. Ele nu trebuie modificate de programator. Cu toate acestea, programatorul trebuie să inspecteze **count_s.hh**, unde găseşte **declarațiile funcțiilor virtuale abstracte ale interfeței Count**. (O soluție mai bună ar fi fost ca declarațiile de funcții ce trebuie implementate de programator să fie furnizate într-un fişier separat, aşa cum VisiBroker face pentru Java). Partea care interesează pentru implementarea serverului are următorul aspect:

```
Class _sk_Counter              // corespunde modulului Counter modelat aici ca o clasa
{ public:
  class _sk_Count: public Counter::Count                 // corespunde interfetei Count
        { virtual CORBA::long sum() = 0;                 //citeste atribut
          virtual void sum (CORBA::long val) = 0;        //scrie atribut
          virtual CORBA::long increment() = 0;           //op. incrementare
        //alte operatii skeleton implementate automat
        . . .
        };
};
```

3.4. Implementarea obiectului server

Orice **server CORBA** trebuie să aibă un **program principal** care inițializează mediul ORB şi porneşte obiectele. În plus, serverul trebuie să prevadă şi **implementări ale interfețelor CORBA** care sunt definite în IDL. În acest exemplu, trebuie implementată o singură **interfață**, **Counter.Count**. Scriem o clasă C++ numita **CountImpl** pentru a implementa această interfață.

Implementarea clasei **CountImpl** se derivează din clasa skeleton corespunzătoare. În acest fel, clasa server moşteneşte funcționalitatea modelului obiectelor CORBA. De asemenea, astfel clasa server obține **funcțiile skeleton** ce permit ORB să invoce automat metodele obiectului (**up-calls**).

Sarcina este, deci, să se implementeze **CountImpl** şi funcția main pentru server.

```
// countimpl.h definiția clasei pentru implementarea lui Count
// VisiBroker pentru C++
# include <count_s.hh>
class CountImpl:public _sk_Counter:: _sk_Count
{ private:
        long _sum;
  public:
        CountImpl (const char * object_name=NULL);
        CORBA::long sum();
        void sum(CORBA::long val);
        CORBA::long increment();
};
```

Definiția clasei **CountImpl** este realizată pornind de la codul generat de compilatorul IDL, la care s-a adăugat un **constructor** al clasei. Ca de obicei, clasa este derivată din skeleton-ul său, _sk_Count.

În ce priveşte implementarea **CountImpl**, ea are următoarea descriere:

```
// countimpl.cpp Count Implementation, VisiBroker pentru C++
# include "countimp.h"
// Constructor
CountImpl::CountImpl (const char *object_name)
                    :_sk_Counter :: _sk_Count (object_name)
{ cout<<"Count object created"<<endl;
  this->_sum=0;
}
// citeste valoarea atributului sum
CORBA::long CountImpl::sum()
{  return this->_sum; }
// scrie valoarea atributului sum
void CountImpl::sum(CORBA::long val)
{  this->sum=val; }
// incrementează suma
CORBA::long CountImpl::increment()
{  this->_sum++;
   return this->_sum;
}
```

Constructorul **CountImpl** apelează părintele (skeleton-ul) pentru a crea un obiect cu nume. Fiecare instanță a clasei CountImpl va avea un nume **persistent**. Dacă se invocă superclasa fără argument, atunci el va crea un obiect **anonim** (sau tranzitoriu).

3.5. Implementarea serverului

Înainte de a continua implementarea serverului, să ne referim la un alt element CORBA care intervine în realizarea apelurilor de operații. Este vorba de adaptoarele de obiecte, **Object Adapters**. Rolul lor este multiplu:

- **Înregistrarea obiectelor** – OA include operații care permit unor entități de limbaj de programare să se înregistreze ca implementări de obiecte CORBA.
- **Generarea referințelor** de obiecte CORBA
- **Activarea procesului server** – dacă este necesar, OA porneşte procesele server în care pot fi activate obiectele
- **Activarea obiectelor** – OA activează obiectele, dacă ele nu sunt deja active la sosirea cererilor.
- **Demultiplexarea cererilor** – OA cooperează cu ORB pentru a asigura ca cererile pot fi primite prin conexiuni multiple, fără blocare nesfârşită a vreunei conexiuni
- **Apeluri de obiecte** (object upcalls) - OA distribuie cererile către obiectele înregistrate.

În mod normal, un OA este necesar pentru fiecare limbaj de programare. De exemplu, un obiect implementat in C s-ar **înregistra** în OS furnizând un pointer de struct care să țină starea obiectelor şi pointerii de funcții corespunzătoare operațiilor obiectului, aşa cum sunt definite de interfețele IDL. Pentru C++, o implementare de obiect poate fi derivată dintr-o clasă de bază standard care include interfața pentru apelurile de operații.

În consecință, CORBA admite mai multe adaptoare dar actualmente prevede una: **Basic Object Adapter (BOA)**. Credința inițială a fost că un singur BOA ar fi suficient. Pentru a suporta mai multe limbaje de programare, specificația acestuia a fost păstrată la un nivel destul de vag în anumite privințe, cum ar fi şi aceea a înregistrării obiectelor. Aceasta a generat probleme de portabilitate a

implementării BOA, fiecare furnizor de ORB "completând" părțile absente cu soluții proprii. Problema portabilității BOA este încă în studiul OMG.

Urmează elaborarea programului principal **server.cpp**.

```
// server.cpp Count server, VisiBroker pentru C++
# include "countimp.h"
int main (int argc, char * const * argv)
{ try
{ // initializare ORB
  CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
  // init BOA
  CORBA:: BOA_ptr boa = orb->BOA_init(argc, argv);
  // creează obiect Count
  Counter::Count_ptr count = new CountImpl ("MyCount");
  // exportă noul obiect – înregistrează cu BOA
  boa->obj_is_ready (count);
  // gata să servească cererile
  boa->impl_is_ready();
} catch (const CORBA::Exception& e)
 { cerr<<e<<endl; exit (-1);}
return (0);
}
```

Programul principal face următoarele operații:

- Inițializează ORB – aceasta se face prin apelul unei funcții din interfața CORBA API şi are efect obținerea unei referințe de pseudo-obiect CORBA::ORB. Un pseudo-obiect este un obiect creat de ORB, dar care poate fi invocat ca orice alt obiect. ORB însuşi este un pseudo-obiect.
- Inițializează BOA - folosind referința orb, inițializează BOA; inițializarea întoarce o referință la BOA.
- crează obiectul **CountImpl**. Aici serverul asociază obiectului un nume (marker) la crearea sa. Numele trebuie să fie unic în cadrul serverului. Dacă un server crează mai multe obiecte **Count**, el trebuie să asigneze acestora nume distincte. Ca alternativă, asocierea mărcilor cu obiecte poate fi lăsată în seama ORB, aplicația având posibilitatea să modifice mărcile ulterior creării obiectelor. **Numele** unui obiect este mai complicat, av\nd mai multe componente: numele serverului, interfața, marker-ul.
- Folosind referința BOA înregistrează în BOA noul obiect creat, CountImpl.
- Anunță BOA că obiectul este gata să lucreze şi că aşteaptă primirea unor invocări de servicii. Funcția **impl_is_ready** nu redă controlul imediat. Ea blochează serverul până la apariția unui eveniment, tratează evenimentul şi re-blochează serverul în aşteptarea unui alt eveniment. Funcția se termină la apariția unui **time-out** (a cărui durată este programabilă) sau a unei excepții.

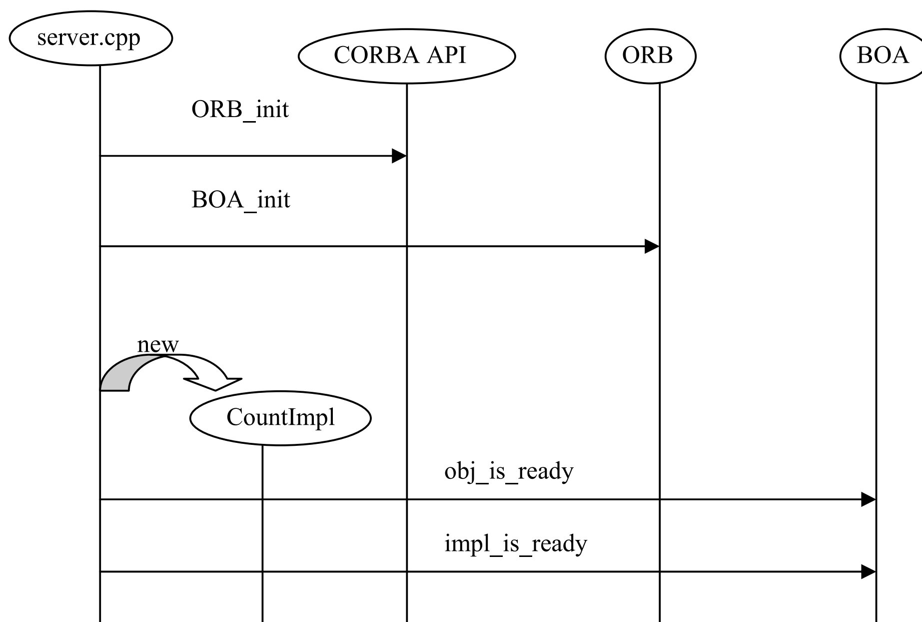

Figura 3.4. Acțiunile programului server

După cum se vede, referințele la obiecte CORBA se pun în corespondență cu tipuri-pointeri în C++: CORBA::ORB_ptr, CORBA::BOA_ptr. Acestea au semantica unor pointeri C++, ceea ce înseamnă (printre altele) că programatorul trebuie să includă în program un CORBA::release() explicit pentru a elibera obiectul.

O altă formă de referință este _var, ca în CORBA::ORB_var, care eliberează automat memoria obiectului referit atunci când referința este distrusă sau folosită pentru un alt obiect.

În altă ordine de idei, este de remarcat modul de tratare a **excepțiilor** în CORBA. Ea se bazează pe mecanismul **try-throw-catch**. Aşa cum se arată şi în exemplu, tratarea excepțiilor este o parte generică a programării client/server, fiind reprezentată în program prin perechea try-catch.

Instrucțiunea **try** are semnificația "încearcă aceste instrucțiuni şi vezi dacă obții o eroare". Ea trebuie urmată de o clauză **catch** care spune "voi trata orice eroare care corespunde argumentului meu".

O **excepție** IDL este tradusă printr-o clasă C++. În exemplul dat, producerea unei excepții de sistem determină afişarea ei şi oprirea programului. Operatorul << definit pe clasa **SystemException** produce o descriere textuală a excepției particulare produse. În ce priveşte transmiterea excepției (declanşarea ei) ea foloseşte instrucțiunea **throw** în C++.

De notat că nu se utilizează o buclă do-forever pentru a servi cererile clienților. CORBA şi BOA furnizează "dedesubt" această funcționalitate. Tot ceea ce se cere programatorului este să se implementeze interfețele şi să se înregistreze în BOA.

3.6. Implementarea clientului

Partea **client** a aplicației constă dintr-un program **main** şi fişierul său antet. Programul trebuie să realizeze următoarele operații:

- Să inițializeze ORB
- Să localizeze un obiect **Count** distant
- Să inițializeze la zero atributul **sum**
- Să calculeze timpul de start
- Să invoce metoda increment de 1000 de ori
- Să calculeze timpul scurs
- Să afişeze rezultatele

```
//Client.cpp Count static client, VisiBroker pentru C++
# include <count_c.hh>
```

```
# include <iosteam.h>
# include <stdlib.h>
# include <time.h>
# include <sys\types.h>
# include <sys\timeb.h>
struct timeb timebuff;
double startTime, stopTime;
int main (int argc, char * const* argv)
{ try
	{
	cout << "initialize ORB" << endl;
	CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
	//bind to the Count Object
	cout << ”Binding to Count Object”<<endl;
	Counter::Count_var Counter=Counter::Count::_bind(“MyCount”);
	/*Counter va contine un pointer la “proxy” intermediarul pentru
	serverul CountImpl */
	Counter->sum((long)0);
	ftime(&timebuff);		//calcul timp de start
	startTime=((double)timebuff.time+((double)timebuff.millitm)/
	(double)1000);
	//increment
	for (int i=0; i<1000; i++)
	{ Counter->increment(); }
	//calcul timp stop
	stopTime==((Double)timebuff.time+((double)timebuff.millitm)/
	((double)1000);
	cout << (stopTime – startTime) <<”nsecs”<endl;
	cout <<”Sum =”<<Counter->sum();
	}
catch(CORBA::SystemException &excep)
	{cout<<”System Exception”<<endl;
	cout <<excep;
	return(1);
	}
  return (0);
}
```

CORBA permite invocarea obiectelor distante folosind semantica uzuală C++. Pentru a invoca metodele, este mai întâi necesară obținerea referinței obiectului, realizată aici cu **bind**. Această metodă este implementată automat de clasa C++ intermediară **Count**. Pentru referință se foloseşte Count_var (în loc de count_ptr), pentru a realiza automat gestiunea memoriei.

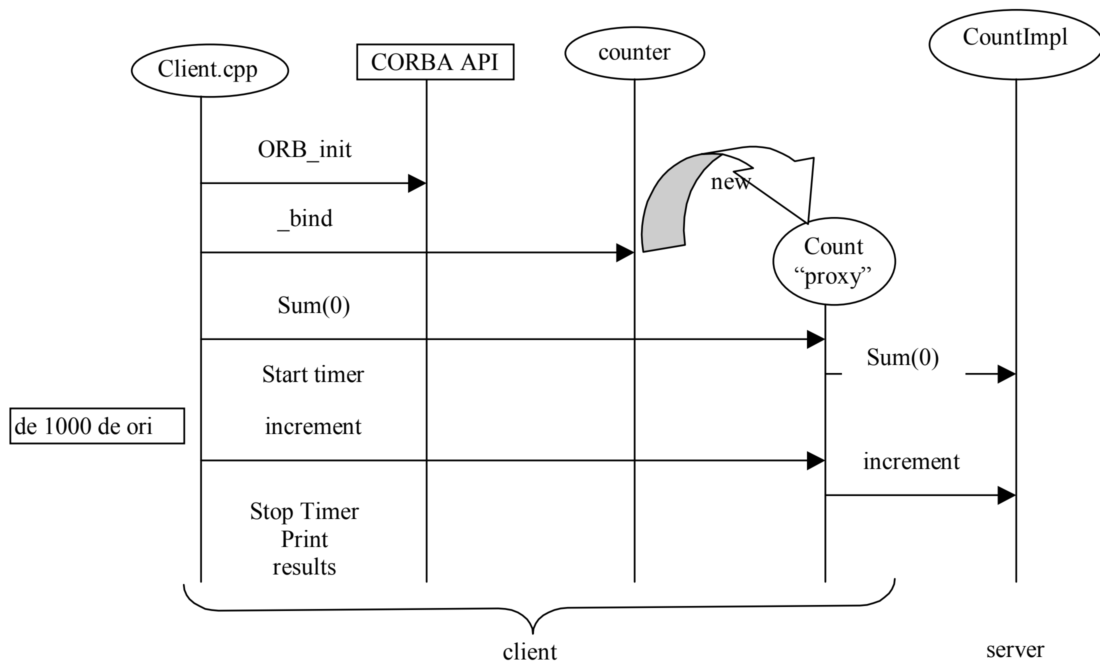

Figura 3.5. Acțiunile programului client

Funcția membru **Count::_bind()** cere ORB să caute un obiect care oferă interfața **Count**. Parametrul “MyCount” este o informație suplimentară pentru ORB, care indică numele (sau marker-ul) obiectului căutat. Marca este parte a referinței obiectului şi este un nume unic în cadrul serverului. Dacă un server crează mai multe obiecte **Count**, el trebuie să dea un nume distinct fiecăruia. Ca alternativă, asocierea mărcilor cu obiectele poate fi lăsată în seama ORB, aplicația având posibilitatea să modifice mărcile ulteriror creării obiectelor. Prin specificarea mărcii, clientul optează pentru un anumit obiect **Count**, în situația în care există mai multe astfel de obiecte într-un server (nu este cazul aplicației de față).

În acest caz, legarea se face la un obiect din afara spațiului de adrese al apelantului. Ca urmare, ORB va construi un “proxy” pentru acest obiect în spațiul de adrese al clientului. Funcția **_bind**() întoarce o referință la obiectul “proxy”. În exemplul dat, referința este asignată unei variabile de tip **Count_var** (care gestionează memoria pentru “proxy”).

Apelul lui **_bind()** nu este singura cale prin care un client poate obține referința unui obiect cu care să comunice. Alte soluții sunt:
- serverul poate înregistra obiectul cu **Naming Service**, asociindu-i un nume; clientul care cunoaşte numele poate cere referința obiectului de la Naming Service;
- un client poate primi o referință de obiect ca rezultat sau ca parmetru **out** al unui apel de operație IDL; aceasta va însemna totodată crearea unui “proxy” în spațiul de adrese al clientului.

4. CORBA dinamic

Exemplele date până în prezent folosesc invocarea statică a serviciilor prin intermediul stub-urilor şi skeleton-urilor precompilate. Un client necesită un stub pentru fiecare interfață pe care o foloseşte. Această cerință devine limitativă în cazul Internetului, unde clienții pot apela, teoretic, la milioane de obiecte din rețea şi unde apar frecvent noi servicii şi interfețe.

Un alt exemplu unde soluția este limitativă este cel al unei **porți (gateway)** între două sisteme de obiecte diferite: CORBA şi un sistem străin. Atunci când primeşte o invocare de la un sistem străin de obiecte, poarta trebuie să o convertească într-o cerere către obiectul CORBA solicitat. Recompilarea programului porții odată cu adăugarea fiecărui nou obiect este o soluție complet nepractică. Din fericire, CORBA suportă două interfețe pentru invocări dinamice:

- Dynamic Invocation Interface (DII) pentru invocările dinamice de cereri ale clienților şi
- Dynamic Skeleton Interface (DSI) care realizează dispecerizările dinamice către obiecte.

DII şi DSI pot fi văzute ca stub-uri si skeleton-uri generice. Fiecare este o interfață prevăzută direct de ORB şi independentă de interfețele OMG IDL ale obiectelor invocate.

Cu DII, un client poate invoca orice operație, a oricărui obiect, fără a avea nevoie de informații despre acestea la momentul compilării. Informațiile necesare sunt "descoperite" în momentul invocării.

4.1. Construirea unei invocări dinamice

Cum **descoperă** clienții obiectele distante? Sunt posibile mai multe mecanisme.

- În cea mai simplă abordare, clientului i se poate da o **referință de obiect**, convertită în şir de caractere. Referințele de obiecte în mediile CORBA sunt robuste, în sensul că ele pot fi memorate persistent într-un fişier, în forma de şir de caractere. (stringified object reference). Referințele pot fi regăsite ulterior şi pot fi folosite pentru invocarea obiectelor, cu condiția ca acestea să mai existe. ORB garantează că referințele rămân valide, chiar dacă se produc erori de rețea ce conduc la pierderea comunicației cu obiectele, caz în care ORB va realiza automat refacerea legăturii. Clientul face conversia de la şir la referință de obiect folosind una din metodele interfeței ORB (string-to-object) şi realizează apoi invocări ale obiectului.
- Clienții pot căuta obiectele după **nume**, folosind Naming Service.
- În fine, pot afla obiectele după **atribute**, folosind Trader Service.
- La acestea adăugăm posibilitatea ca un client să obțină o referință de obiect ca rezultat sau parametru **out** al unei invocări anterioare.

Odată dispunând de referința obiectului, clientul o poate folosi pentru a regăsi **interfața** obiectului şi a realiza construcția dinamică a cererii. În cerere, trebuie specificată metoda invocată şi parametrii corespunzători. Aceste informații sunt obținute uzuala din depozitul de interfețe – Interface Repository (IR). În consecință, construcția dinamică a unei invocări reclamă etapele descrise în continuare.

(1) Obținerea numelui interfeței.

Odată ce dispunem de o referință la obiectul server, putem cere acestuia numele interfeței sale. Pentru asta se invocă metoda **get_interface** (figura 4.1). Acest apel întoarce o referință la un obiect **InterfaceDef** din IR, care descrie interfața necesară clientului.

(2) Obținerea descrierii metodei, din IR.

Putem folosi **InterfaceDef** ca un punct de intrare pentru navigarea în Interface Repository, IR. Putem obține astfel o mulțime de detalii despre interfață şi despre metodele sale. CORBA specifică aproape zece apeluri pentru inspectarea IR. De exemplu, clientul poate apela **lookup_name** pentru a găsi metoda pe care vrea să o invoce (de fapt o referință la un obiect **OperationDef** care descrie operația), iar apoi **describe** pentru a obține definiția IDL completă a metodei.

Ca alternativă, se poate apela **describe_interface** pentru a obține o definiție completă a interfeței şi pentru a găsi metoda care trebuie invocată.

(3) Crearea listei de argumente.

(3a) CORBA specifică o structură de date autodefinită pentru transmiterea parametrilor, anume *Named Value List*. Pentru implementarea acestei liste se foloseşte un pseudo-obiect **NVList**. Pentru crearea listei se invocă **create_list** şi apoi **add_item**, în mod repetat, pentru a adăuga listei fiecare argument.

(3b) Ca alternativă, clientul poate invoca **create_operation_list** pe un obiect CORBA::ORB, ca urmare a căreia lista este creată de ORB. Acestei metode trebuie să i se dea numele operației pentru care trebuie creată lista de argumente.

(4) Crearea cererii.

O cerere este un pseudo-obiect CORBA care conține numele metodei, lista de argumente şi valoarea returnată ca rezultat.

(4a) Pentru aceasta se invocă **create_request**, căreia i se transmite numele metodei, NVList şi un pointer la valoarea returnată.

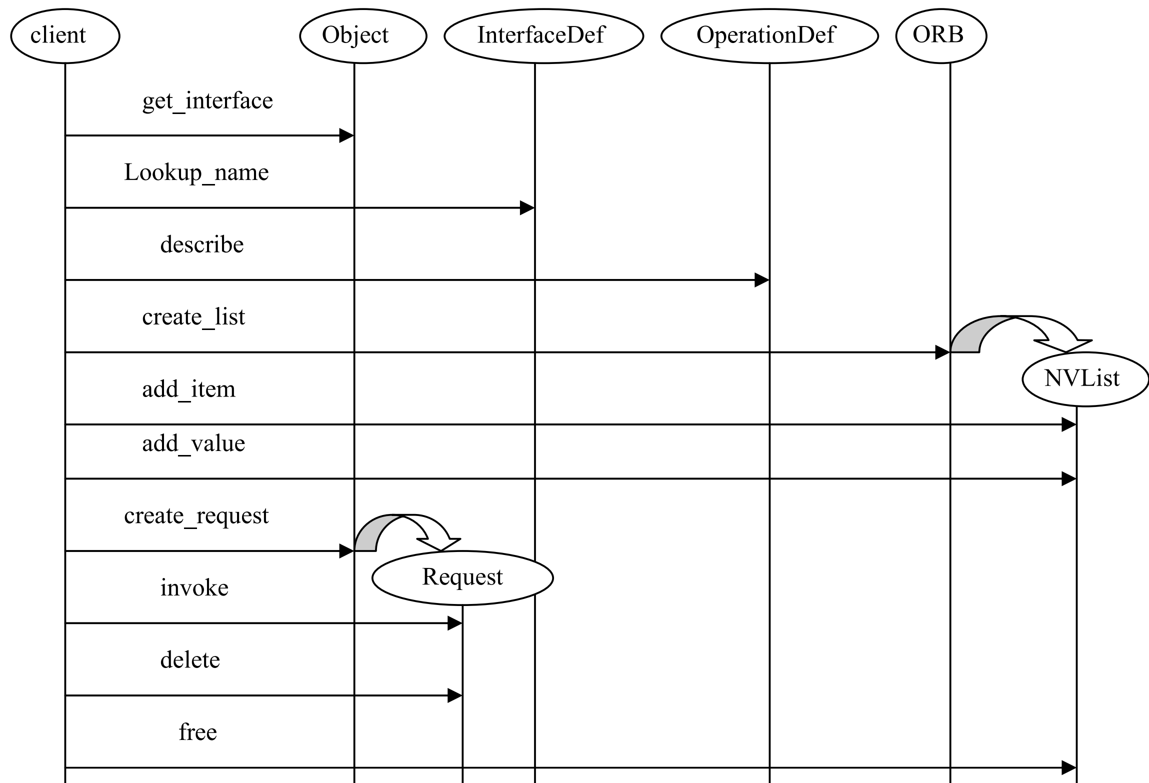

Figura 4.1. Construuirea unui apel dinamic

(4b) Ca alternativă, se poate crea o versiune scurtă a cererii, apelând **request**, căreia i se pasează doar numele metodei. Soluția se foloseşte pentru metodele fără parametri.

(5) Invocarea cererii.

Invocarea cererii se poate face printr-una din următoarele trei metode:

- Apelând **invoke**; se transmite cererea şi clientul se blochează până se obțin rezultatele; această formă se numeşte invocare sincronă
- Apelând **send_deferred**; clientul invocă cererea şi continuă prelucrarea în timp ce cererea şi continuă prelucrarea în timp ce cererea este dată serverului şi tratată; clientul trebuie să colecteze ulterior răspunsul apelând **poll_response** sau **get_response**; această formă se numeşte invocare sincronă amânată.
- Clientul invocă cererea şi continuă apoi prelucrarea; nu există răspuns, deci apelul este definit ca o datagramă; pentru asta se apelează **send_oneway**.

4.2. Interfețele de invocare dinamică

Pentru invocarea dinamică a metodelor se folosesc **servicii din nucleul CORBA**. Serviciile sunt dispersate în patru interfețe din modulul CORBA:

CORBA::Object. Este o interfață de **pseudo-obiect** care defineşte operațiile ce trebuie suportate de orice obiect CORBA. Aceste operații sunt executate de ORB şi sunt moştenite la crearea unui obiect. Interfața include trei metode folosite în invocarea dinamică:

- get_interface - obține o referință la un obiect din IR care descrie interfața
- create_request - crează un obiect **Request**
- _request - crează o versiune "scurtă" a cererii

CORBA::Request este o interfață de pseudo-obiect care defineşte operațiile asupra unui obiect distant. Sunt incluse aic:

- add_arg - adaugă un argument cererii
- invoke - apel sincron
- send_oneway - apel datagramă
- send_deffered - apel sincron întârziat
- get_response - aşteaptă răspunsul
- poll_response - testează sosirea răspunsului
- delete - şterge obiectul **Request** din memorie

CORBA::NVList este o interfaţă de pseudo-obiect care facilitează construirea listei de parametri:

- add_item - adaugă un parametru
- add_value - setează valoarea parametrului
- get_count - află numărul total de parametri alocaţi în listă
- remove - şterge un element din listă
- free - şterge structura de listă
- free_memory - eliberează memoria alocată dinamic argumentelor **out**

CORBA::ORB este o interfaţă de pseudo-obiect ce defineşte metode ORB generale. Şase metode sunt specifice construcţiei dinamice a cererilor:

- create_list - crează o listă vidă ce urmează a fi populată
- create_operation_list - crează o listă iniţializată de ORB
- send_multiple_requests_oneway - trimite cereri datagramă multiple
- send_multiple_requests_deferred - trimite cereri întârziate multiple
- poll_next_response - tesetază următorul răspuns
- get_reset_response - aşteaptă următorul răspuns

În afara acestor interfeţe, pentru a construi o invocare se mai folosesc obiecte din Interface Repository.

4.3. Dynamic Skeleton Interface

Analogul DII de partea serverului este DSI. Aşa cum DII permite clienţilor să invoce cereri, fără a fi necesare stub-uri statice, DSI permit ca serverele să fie scrise fără a avea skeleton-uri compilate static în program.

Pentru aceasta, serverul poate defini o **funcţie** care va fi informată de orice invocare de operaţie sau atribut şi care:

- determină identitatea obiectului invocat
- determină numele operaţiei, tipurile şi valorile argumentelor
- îndeplineşte acţiunea cerută de client
- construieşte şi întoarce rezultatul.

Principala sa utilizare este construcţia porţilor (gateways) între sisteme de obiecte CORBA şi non-CORBA. Iniţial, DSI a fost gândit pentru a fi utilizat în porţile dintre diferite ORB se folosesc protocoale diferite. Odată cu adaptarea IIOP această funcţie s-a dovedit inutilă.

5. Iniţializarea

Să ne reamintim din exemplul de invocare statică a metodelor de obiecte-server că în CORBA 2.0 sunt definite metode de **iniţializare** pe care orice ORB trebuie să le furnizeze pentru a permite unui obiect să se "integreze" într-un mediu distribuit. Aceste metode sunt implementate de pseudo-obiectul CORBA::ORB.

Un **pseudo_obiect** este un obiect creat direct de ORB, dar care poate fi invocat ca oricare alt obiect. ORB însuşi este un pseudo-obiect.

Trei metode sunt specifice iniţializării unui obiect şi sunt disponibile după execuţia unui apel ORB_init:

- BOA_init
- list_initial_services
- resolve_initial_references

Un **scenariu** posibil de inițializare include următoarele faze:

- Obținerea unei referințe de obiect pentru propriul ORB. Apelul CORBA API, **ORB_init** permite unui obiect să informeze ORB despre existența sa şi să obțină o referință la un pseudo-obiect ORB. De accentuat că **ORB_init** este un apel API şi nu o invocare de metodă.
- Obținerea unui pointer la BOA.
- Invocarea metodei **BOA_init** pe obiectul **ORB** permite obiectului să informeze adaptorul despre existența sa şi să obțină referința pseudo-obiectului BOA.
- Descoperirea serviciilor inițiale disponibile.
- Invocarea metodei **list_initial_services** pe pseudo-obiectul ORB permite obținerea unei liste cu numele obiectelor cunoscute, ca de exemplu, Interface Repository, Trader Service si Naming Service. Acestea sunt returnate ca o listă de referințe convertite la şiruri de caractere.
- Obținerea referințelor de obiecte pentru serviciile dorite. Prin invocarea metodei **resove_initial_references** se obțin referințele de obiecte pentru serviciile dorite, din referințele convertite la şiruri de caractere.

Serviciul de inițializare este un fel de Naming Service elementar. El permite obținerea unei liste de servicii binecunoscute: Naming Service, Trading Service, precum şi alte servicii de navigare, căutare, agenți, etc. Utilizarea acestora permite găsirea altor obiecte din universul ORB.

5.1. Serviciul de nume (Naming Service)

Naming Service păstrează p bază de date de legături (bindings) între nume şi referințe de obiecte. Serviciul are operații pentru:

- rezolvarea unui nume
- crearea, ştergerea, listarea unor legături.

Un nume este o secvență IDL de componente de nume. O componentă este o structură de două şiruri: primul este numele componentei; al doilea este un atribut care nu este interpretat de Naming Service, fiind destinat utilizării de către aplicație. De exemplu, el poate preciza că numele trebuie interpretat ca numele unui catalog, sau al unui disc.

Fiecare componentă, cu excepția ultimeia, defineşte un NamingContext (similar unui director într-un sistem de fişiere). Aceasta conferă sistemului de nume o structură ierarhică ce ordonează căutarea pentru rezolvarea numelor: prima componentă dă numele unui context în care se caută al doilea nume; procesul continuă până la ultima componentă.

5.2. Serviciul de Trading

Față de această funcționare simplă, un **Trader** este mult mai complex. Oricum, şi funcția sa este mai complexă, Trader-ul permițând căutarea obiectului cel mai potrivit într-un set larg de obiecte similare.

Cum funcționează un serviciu **Trader**? **Exportatorii** sau furnizorii de servicii anunță Trader-ul despre serviciile oferite. **Importatorii** sau consumatorii de servicii folosesc Trader-ul pentru a găsi serviciile care satisfac nevoile lor.

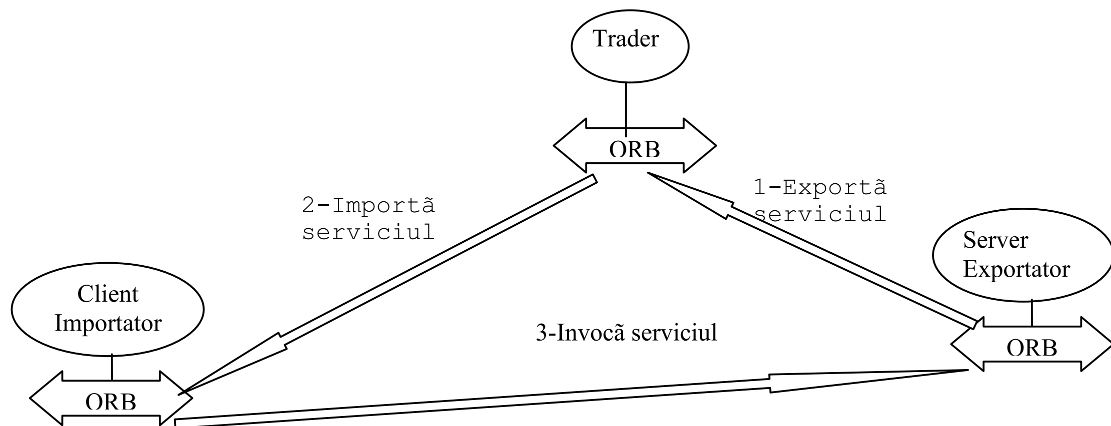

Figura 5.1. Funcționarea unui Trader

Mai întâi, un nou furnizor de servicii înregistrează serviciile la trader, căruia îi furnizează următoarele informații relevante:

- **O referință de obiect** pe care clienții o vor folosi pentru a se conecta la serviciu şi a invoca operațiile. Ea reprezintă referința de obiect a interfeței care furnizează serviciul.
- **Tipul serviciului** care include informații asupra numelor operațiilor (sau metodelor) apelabile împreună cu tipurile parametrilor şi rezultatului.
- **Propritățile serviciului** care sunt perechi nume-valoare care definesc oferta. Ele descriu capabilitățile serviciului şi se plasează în două categorii: obligatorii şi opționale.

Trader-ul păstrează un **depozit de tipuri de servicii**. Putem avea, de exemplu, un tip **restaurant**, pentru care proprietățile serviciului pot fi: meniul, specialitățile, adresa, numărul de locuri, orarul, etc.

Trader-ul păstrează descrierile în **ServiceTypeRepository**. Totodată, păstrează o bază de date de obiecte server, care sunt instanțe ale acestor tipuri. Clienții pot intra în contact cu Trader-ul cerându-i să găsească serviciile care se potrivesc cel mai bine unui anumit set de cerințe. Un serviciu care ar corespunde cererii trebuie să aibă un **tip** care se potriveşte cu cererea clientului şi proprietățile care satisfac criteriile impuse de client.

Descrierea criteriilor se face într-un limbaj de constrângeri ce precizează:

- tipurile de proprietăți ce pot apare în cosntrângeri (de exemplu, întregi, reali, char etc.)
- operatorii admişi (comparații, apartenența la o secvență, existența unei proprietăți etc.)

Clientul poate preciza **preferințele** asupra ordinii în care trader-ul ar trebui să furnizeze **ofertele** care se potrivesc cererii (de exemplu, mai întâi oferta care are un anumit parametru, sau o valoare maximă pentru un parametru, sau o ordine oarecare, sau ordinea în care ofertele sunt găsite de trader). Clientul poate specifica şi o **politică** pe care să o urmeze trader-ul şi care fixează elemente ca număruil maxim de oferte furnizate sau amploarea căutării.

Trader-i din domenii diferite pot crea **federații**. Un Trader poate oferi serviciile de care răspunde, dar şi servicii ale trader-ilor cu care este în federație.

În scenariul prezentat în continuare intervin următoarele interefțe:

- Lookup - permite clienților şi trader-ilor să descopere şi să importe servicii; are o singură operație, **query**;
- Register - folosită de furnizorii de servicii pentru a anunța serviciile lor;
- ServiceTypeRepository - depozitul tipurilor de servicii.

Scenariul cuprinde următoarele etape:

1. serverul crează un nou tip de serviciu cu **add_type**
2. serverul avertizează Trader-ul despre serviciul său, invocând **export**; el comunică numele tipului de serviciu, referința de obiect care va servi cererea, valorile parametrilor.

3. clientul obține o listă de tipuri de seervicii existente în depozit (list_types)
4. clientul obțşine descrierea unui anumit tip (describe_type); descrierea include parametrii tipului şi interfața de obiect
5. clientul invocă **query** pentru a obține o listă a obiectelor care pot furniza acest serviciu
6. clientul invocă serviciul.

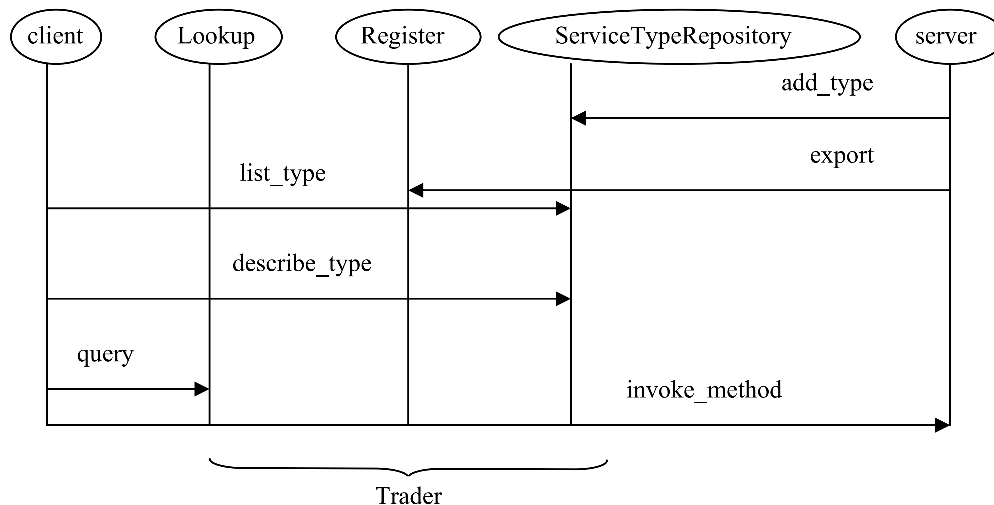

Figura 5.2. Scenariu de funcționare a serviciului Trading

6. Activarea obiectelor

În exemplul dat, la discutarea invocării statice a seviciilor, obiectele **count** trebuiau pornite manual înainte de a servi cererile clienților. Cu excepția cazului creării unui nou obiect, care este implicit pornit de CORBA, **standardul nu prevede o comandă explicită de pornire a unui obiect server**. Filozofia CORBA este de a păstra clientul cât mai simplu: acesta trebuie să "vadă" obiectele server ca fiind întotdeauna pregătite să accepte invocările operațiilor lor. Ca urmare, ORB trebuie să fie capabil să pornească în avans un obiect sau să-l pornească la cerere, atunci când un client invocă obiectul. Aceasta presupune că **serverului i se adaugă o parte de cod care cooperează cu ORB** la pornirea sau oprirea lor.

În principiu, **interfața CORBA::BOA** este folosită pentru a crea şi distruge referințe de obiecte şi pentru a afla ori actualiza informația pe care BOA o păstrează despre referințele la obiecte. BOA **păstrează evidența** obiectelor active şi a implementărilor pe care le controlează. Interfața BOA este folosită pentru a avea acces la această evidență, pentru a afla sau adăuga informații despre obiecte. Iată o scurtă descriere a metodelor CORBA::BOA.

- **create** este invocată pentru a descrie implementarea unei noi instanțe de obiect şi a obține o **referință de obiect** pentru ea. ORB i se pasează mai multe informații:
    - ♦ un nume de interfață care este descrisă în Interface Repository
    - ♦ un nume de implementare care este descrisă în Implementation Repository
    - ♦ un identificator unic (nu este folosit de ORB ci este specifică implementării şi permite diferențierea obiectelor sau specificarea unor identificatori persistenți – Persistent ID).
- **change_implementation** permite actualizarea implementării asociate cu un obiect existent.
- **get_id** permite obținerea identificatorului asociat cu obiectul.
- **dispose** distruge referința obiectului. Separat trebuie distruse resursele obiectului.

Există şi alte metode care permit activarea şi dezactivarea implementărilor şi a obiectelor care rulează în aceste implementări. CORBA cere ca următoarele funcţii să fie disponibile într-o implementare BOA:

- Un Implementation Repository care permite instalarea şi înregistrarea implementării unui obiect
- **Mecanisme** pentru generarea şi interpretarea referinţelor de obiecte, activarea şi dezactivarea implementărilor de obiecte, invocarea metodelor şi pasarea parametrilor.
- Activarea şi dezactivarea obiectelor implementării
- Invocarea metodelor prin skeleton.

CORBA face o distincţie clară între un server şi obiectele sale. Un **server** este un proces, o unitate de execuţie. Un **obiect** implementează o interfaţă. Un server poate conţine unul sau mai multe obiecte, **eventual de clase diferite**. La cealaltă extremă se află un server care conţine codul pentru implementarea unei singure metode. În toate cazurile, obiectele sunt activate în serverele lor. CORBA defineşte patru politici de activare: server partajat (shared server), server ne-partajat, server-per-metodă şi server persistent.

6.1. Server partajat

Toate obiectele cu acelaşi nume de server rezidă în acelaşi proces. BOA activează serverul la prima invocare de metodă a unuia din aceste obiecte. După ce serverul s-a iniţializat, el anunţă BOA că poate prelucra cereri prin apelul **impl_is_ready**. Toate cererile ulterioare (de metode ale obiectelor acestui server) sunt livrate acestui proces. BOA nu activează un alt proces server pentru această implementare. (Când un alt client execută un obiect, legarea se face la aceeaşi copie de server).

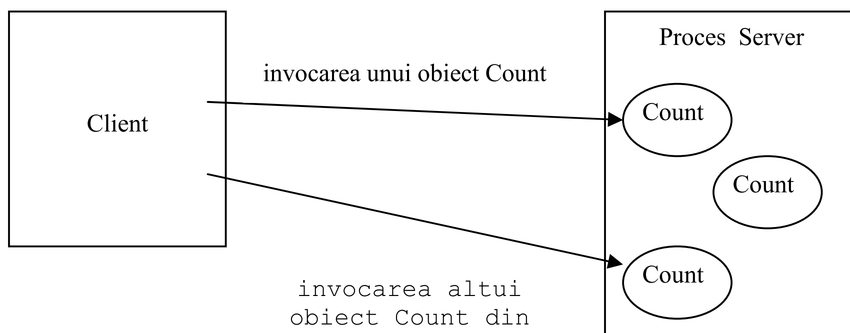

Figura 6.1. Server partajat

În detaliu, lucrurile se petrec după scenariul următor:

1) Serverul creează **instanţe** de obiecte, fie prin invocarea unei fabrici de obiecte CORBA, fie printr-un constructor (Java, C++, …)
2) Noile obiecte se înregistrează în BOA. Dacă obiectul este **nou**, el invocă **BOA::create** care întoarce o referinţă de obiect. Această referinţă este folosită într-un apel **BOA::Object_is_ready**, anuntând OBJ că obiectul este pregătit. Dacă obiectul **există** deja, el are starea înregistrată undeva într-o memorie permanentă. Folosind referinţa la obiect (care există deja!) se poate obţine identificatorul unic (**get_id**) şi apoi **PersistentID** asociat cu obiectul. Pe baza acestuia se regăseşte şi încarcă starea obiectului. Obiectul invocă apoi **obj_is_ready** anuntând ORB că e pregătit.
3) După ce a pornit toate obiectele, serverul invocă **impl_is_ready**, anuntând că este gata să accepte invocări de la clienţi.
4) Când un obiect nu mai este referit, el poate fi dezactivat prin **deactivate_obj**.
5) Când un proces server se termină, el anunţă BOA printr-un apel **deactivate_impl**.

6.2. Server nepartajat

Fiecare obiect rezidă într-un proces separat. La prima invocare a unui obiect este pornit procesul corespunzător. Când obiectul a încheiat faza de inițializare, el anunță BOA prin **obj_is_ready**. Obiectul rămâne activ până la un apel **dezactivate_obj**. Ori de câte ori se apelează un obiect care nu este activ se pornește un alt obiect cu aceeași implementare. Acest mod se folosește:

- atunci când obiectele necesită cantități mari de resurse
- când se dorește mărirea gradului de paralelism (ca alternativă la thread-uri).

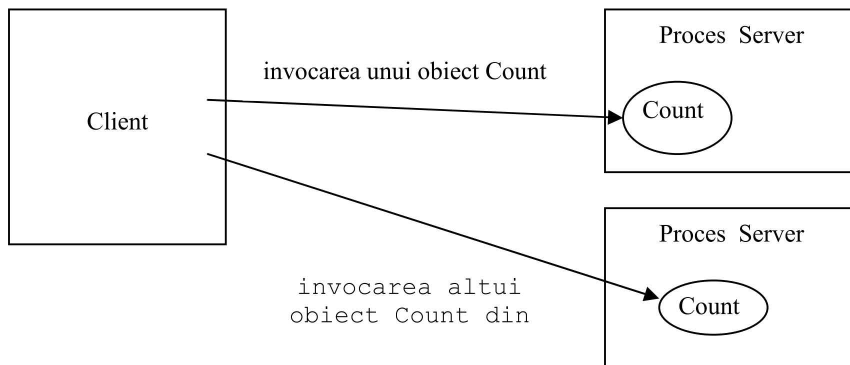

Figura 6.2. Server nepartajat

6.3. Server-per-metodă

Un nou server este activat odată cu fiecare cerere și dezactivat odată cu satisfacerea cererii. Nu este deci nevoie ca implementarea să anunțe BOA când un obiect este activat/dezactivat.

6.4. Server persistent

Serverul este activat prin mijloace din afara adaptorului BOA. Odată activat, serverul anunță BOA, printr-un apel **impl_is_ready**, că poate primi cereri de la clienți fiind tratat în continuare ca un server partajat.

6.5. Concluzii

După cum se vede, rolul BOA în păstrarea evidenței obiectelor active sau activabile la cerere este esențial. BOA folosește aceste informații pentru a putea realiza dispecerizarea cererilor clienților.

Pentru actualizarea evidenței, BOA cere tuturor obiectelor server să se autodescrie și să-și înregistreze permanent starea. În principiu, fiecare obiect își anunță pornirea prin **obj_is_ready**. Când toate obiectele gestionate de un proces server sunt active, acesta apelează **impl_is_ready**.

Ce se întâmplă dacă un proces are foarte multe obiecte? Instanțierea tuturor obiectelor în memorie, la pornire ar putea fi prea costisitoare. Este mult mai eficient să se facă instanțierea unui obiect doar atunci când un client are nevoie de el. **VisiBroker** folosește o versiune modificată pentru **obj_is_ready**, transmițând ca argument un **Activator** pentru obiect. Acesta este o clasă specială care implementează două metode: **activate** și **deactivate**. Prima este apelată de BOA când obiectul este invocat. A doua este apelată când serverul își termină execuția.

7. Păstrarea descrierilor interfețelor

CORBA se deosebește de sistemele client-server tradiționale prin faptul că este **auto-descriptibil**, **dinamic** și **reconfigurabil**. IDL este limbajul metadatelor CORBA, iar IR (Interface Repository) este depozitul metadatelor CORBA, o bază de date care conține specificațiile de interfață ale oricărui obiect recunoscut de CORBA. Aceste elemente permit descrierea consistentă a tuturor seviciilor, componentelor și datelor disponibile. Astfel, componente dezvoltate independent se pot descoperi

reciproc în mod dinamic şi pot coopera. Nu este deci necesar ca programul unui client să includă apeluri la servere particulare, stabilite la compilare şi nemodificabile ulterior. În plus, metadatele disponibile pot fi folosite de instrumente de creare şi gestiune a diferitelor componente de aplicaţii.

IDL este un limbaj declarativ care permite specificarea interfeţelor unor componente cu diverşi clienţi. IR conţine metadate identice cu descrierile IDL, dar în forma compilată. CORBA defineşte **coduri de tip** care reprezintă diverse tipuri de date definite în IDL. Codurile sunt folosite pentru a crea **structuri auto-descrise** ce pot fi comunicate prin sisteme de operare, ORB-uri si IR_uri. Codurile sunt **utilizate**:

- de IR pentru a crea descrieri IDL independente de ORB
- de DII (Interfeţel.e de invocare dinamică) pentru a indica tipurile diferitelor argumente
- de protocoalele Inter-ORB pentru a descrie câmpurile mesajelor communicate între ORB-uri
- de tipul **any** pentru a furniza un parametru generic auto-descris. **any** se foloseşte pentru un parametru al cărui tip nu este fixat la compilare. La execuţie, pentru un parametru **any** se poate transmite o valoare de orice tip. Obiectul ţintă care primeşte un **any** poate obţine **TypeCode**-ul său şi din el poate determina tipul valorii transmise. Concret, clasa CORBA::Any are o funcţie membră **type()** care întoarce o valoare de tip CORBA::TypeCode_ptr. Această valoare poate fi inspectată la execuţie pentru a se afla tipul valorii transmise ca **any**.

Interfaţa CORBA **TypeCode** defineşte un set de metode ce permit operarea cu coduri de tip, compararea lor, obţinerea descriilor, etc.

Fiecare cod de tip are un identificator global unic – **Repository ID** care poate fi folosit într-un spaţiu de nume distribuit. Asocierea dintre cod şi identificator se face la compilarea descrierilor IDL, sau la integrarea lor în IR folosind alte instrumente. Un Repository ID este un şir de caractere cu trei componente, separate de "**:**" reflectând o ierarhie de nume cu trei niveluri. Forma este standardizată şi are o structură foarte generală.

Prima componentă identifică formatul şi poate fi IDL sau DCE (doar aceste două formate sunt definite În CORBA 2.0).

Pentru IDL, a doua componentă este o listă de identificatori despărţiţi prin /. Primul este un **prefix unic**, reprezentând numele unei organizaţii, un nume Internet, etc. Celelalte sunt identificatori IDL care alcătuiesc împreună numele (complet al) tipului. De exemplu, interfaţa **Itrf** a modulului **Mdl** are numele Mdl/Interf.

A treia componentă este o pereche de numere de versiune majoră şi minoră, despărţite prin punct, v_majora.v_minora

7.1. Interface Repository

Interface Repository, IR este o **bază de date de definiţii de obiecte**, generate de un compilator IDL sau introduse prin funcţiile de scriere specifice IR. Obiectele din IR sunt versiuni compilate ale informaţiei care se află în sursele IDL. Altfel spus, pentru fiecare definiţie IDL găsim în IR un obiect care încapsulează descrierea corespunzătoare. Specificaţia CORBA se referă explicit doar la modul în care informaţia din IR este organizată şi poate fi regăsită. Obiectele din IR suportă interfeţe IDL care reflectă construcţia IDL pe care o descrie: ModuleDef, InterfaceDef, AttributeDef, etc. În plus, interfaţa **Repository** serveşte ca rădăcină pentru toate celelalte. Fiecare IR este reprezentat de un obiect Repository. Figura 7.1 arată ierarhia de interfeţe din punct de vedere al conţinutului obiectelor care suportă aceste interfeţe.

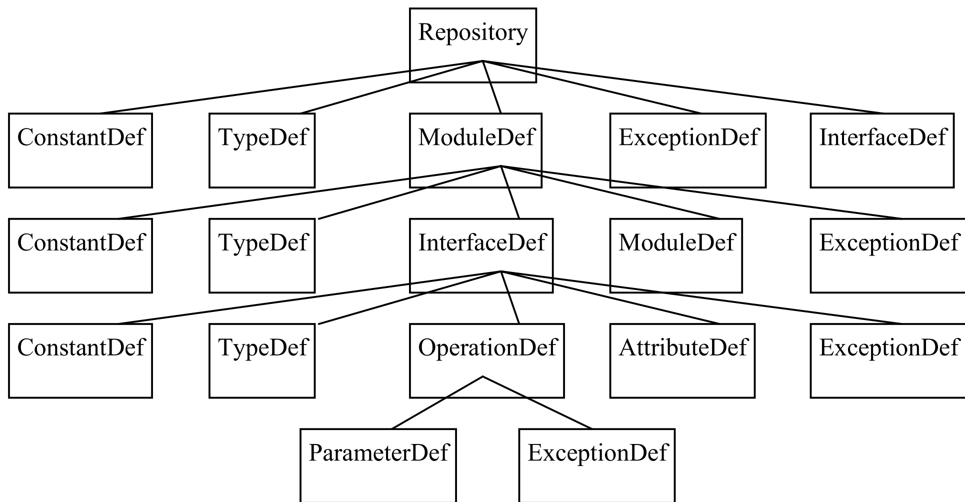

Figura 7.1. Ierarhia de clase

Obiectele corespunzătoare acestor tipuri se grupează după cum:

- sunt **containere** de alte obiecte (Repository)
- sunt **conținute** de alte obiecte (ConstantDef)
- sunt atât **containere** cât şi **conținute** (ModuleDef).

Pornind de la această relație, CORBA stabileşte ierarhia de moştenire pentru tipurile IR, introducând trei interfețe abstracte (interfețe ce nu pot fi instanțiate): **IRObject**, **Contained** si **Container**. Această ierarhie este reprezentată în figura 7.2.

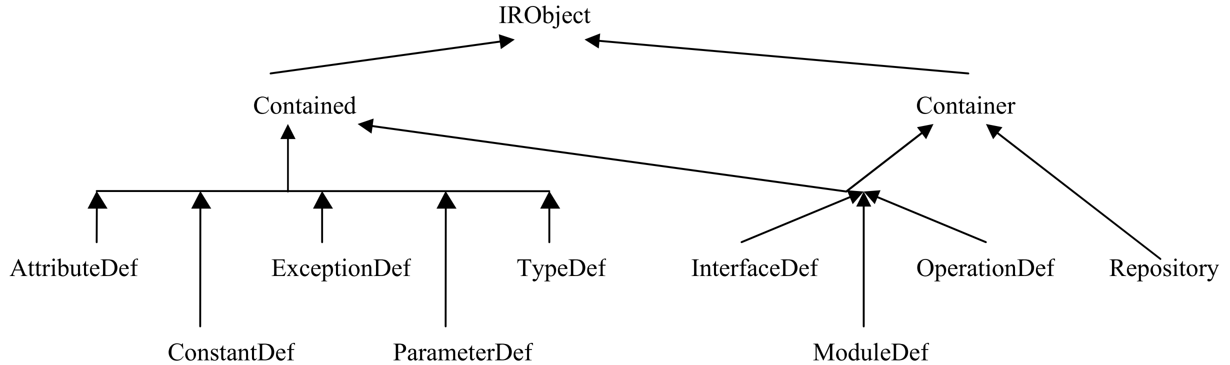

Figura 7.2. Ierarhia IRObject

Obiectele care conțin alte obiecte moştenesc operațiile de navigare din **container**. Obiectele conținute moştenesc comportarea comună din **contained**. Toate obiectele moştenesc din **IRObject**.

Interfețele IR definesc operații care permit citirea, scrierea şi distrugerea metadatelor păstrate în IR. Folosind doar 9 metode, se poate naviga în IR şi se pot extrage informațiile de descriere a obiectelor căutate.

De exemplu, invocarea metodei **contents** a unui obiect **container** întoarce lista obiectelor conținute direct sau moştenite de obiectul respectiv. Folosind această metodă se poate naviga printr-o ierarhie de obiecte. Metoda **lookup-name** aplicată unui obiect **container** permite localizarea unui obiect după nume.

Metoda **lookup_id** aplicată unui obiect **Repository** permite căutarea unui obiect într-un depozit, cunoscând identificatorul său, Repository ID.

8. Protocoale Inter-ORB

Înainte de CORBA 2.0, produsele ORB comerciale nu puteau inter-opera. CORBA 2.0 introduce conceptul de inter-operabilitate şi defineşte două moduri de inter-operabilitate: cea directă şi cea bazată pe o punte. Inter-operabilitatea directă este posibilă între două ORB-uri din acelaşi domeniu: care înțeleg aceleaşi referințe de obiecte, acelaşi sistem de tipuri IDL şi care partajează aceeşi informaţie de securitate. Inter-operabilitatea bazată pe o punte (bridge) se foloseşte atunci când două ORB-uri din domenii diferite trebuie să coopereze.

Inter-operabilitatea se bazează pe un protocol general: GIUP – General Inter-ORB Protocol, care specifică sintaxa de transfer şi un set standard de formate de mesaje pentru inter-operarea ORB peste o legătură de transport orientată pe conexiuni. IIOP – Internet Inter-ORB Protocol descrie construcția GIOP pe legături de transport TCP/IP.

9. Cîteva servicii CORBA importante

9.1. Serviciul ciclului de viață

9.2. Serviciul de evenimente

9.3. Serviciul de securitate

10. Implementări ORB

Java ORB este un ORB scris în întregime în Java. Cu Java ORB, un applet obişnuit poate invoca metode ale obiectelor CORBA folosind IIOP. Appletul ocoleşte complet CGI şi HTTP, între client şi server stabilindu-se o legătură directă. Trei dintre cele mai cunoscute Java ORB sunt:

- Joe de la Sun
- OrbixWeb de la Iona
- VisiBroker for Java de la Visigenic/Netscape.

10.1. Joe

Produsul NEO de la Sun include:

- Solaris NEO – mediu ce conține suportul de execuție necesar pentru aplicațiile NEO/JOE
- Solstice NEO – instrumente de gestiune a obiectelor în sisteme distribuite
- NEOworks – instrumente de dezvoltare a aplicațiilor (incluzând compilatorul IDL, un depanator, etc).

Joe este un Java ORB pentru clienți. El poate fi încărcat odată cu un applet Java sau poate fi instalat permanent pe maşina client. Joe include un compilator IDL-TO-Java care generează automat stub-uri Java din descrieri IDL. În prezent, obiectele server trebuie scrise pentru platforma NEO, care suportă C şi C++. Versiunea IIOP pentru Joe va fi capabilă să comunice cu orice Java ORB care suportă IIOP.

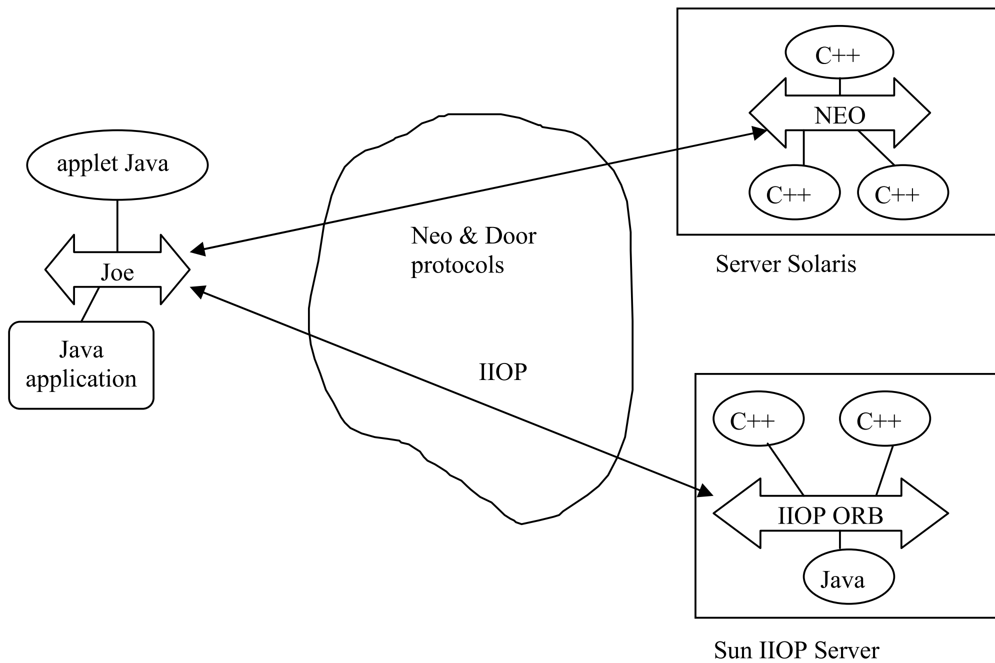

Figura 10.1. Joe

10.2. Orbix

Iona este liderul furnizorilor de tehnologie CORBA. Produsul său Orbix ORB este executat pe 20 de sisteme de operare (Unix, OS/2, NT, Windows 95, Macintosh, VMS).

OrbixWeb V1 este o implementare Java pentru clienți; care permite applet-urilor şi aplicațiilor Java să comunice cu servere Orbix folosind fie protocolul IIOP, fie protocolul Orbix (Iona).

În prezent, obiectele server trebuie scrise în C++. Oricum, OrbixWeb V2 va permite elaborarea lor în Java.

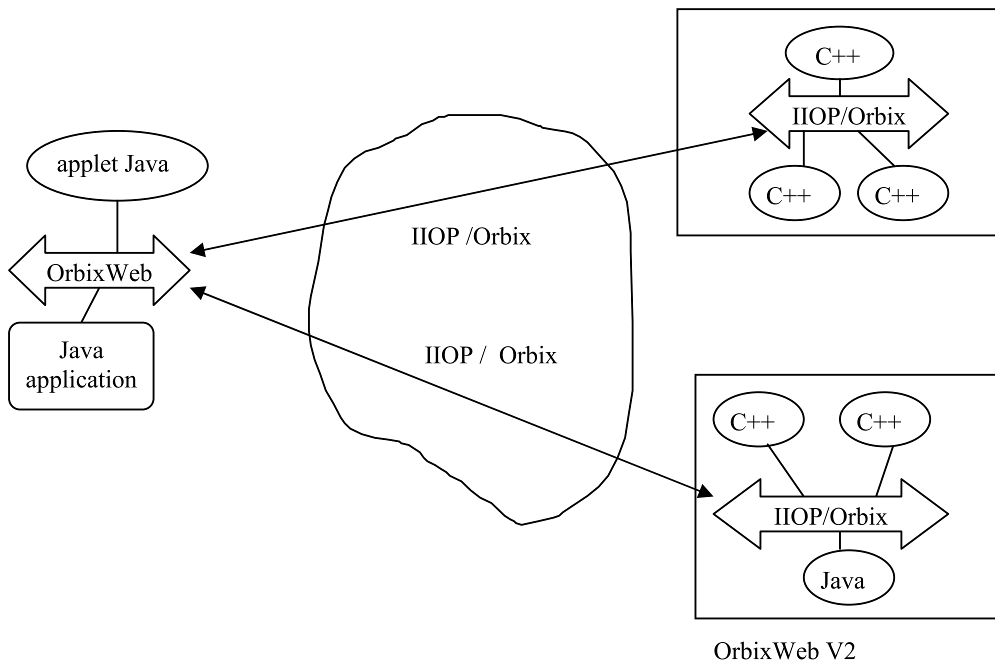

Figura 10.2. OrbixWeb

10.3. VisiBroker

Are două variante: VisiBroker for C++ si for Java.

VisiBroker for Java este un ORB client şi server scris în Java.

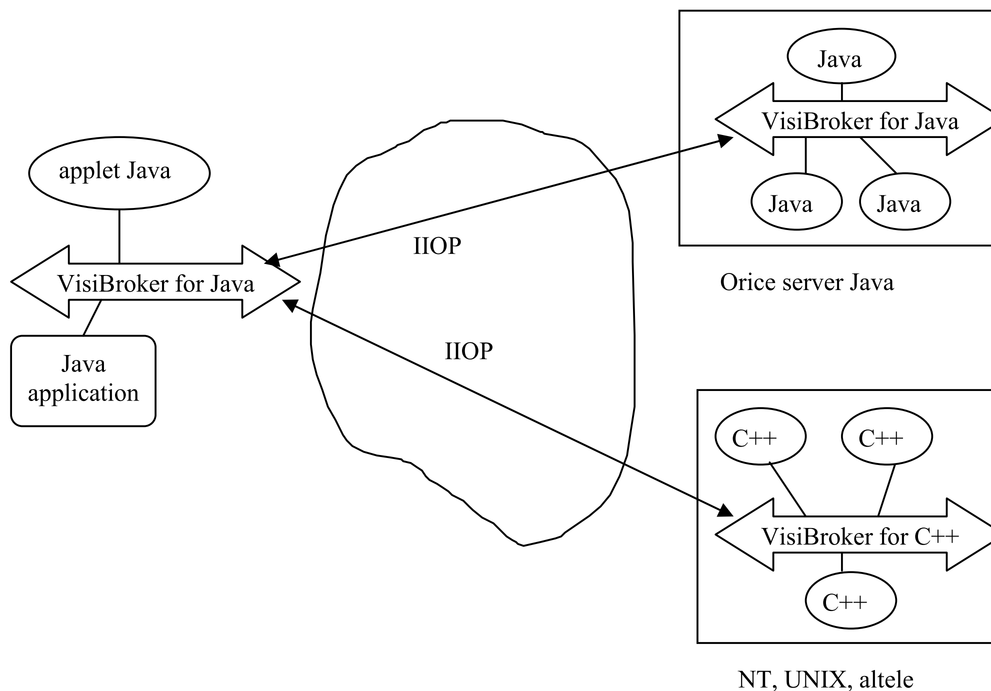

Figura 10.3. VisiBroker

El implementează protocolul IIOP, care permite ca obiectele C++ să apeleze metode Java şi reciproc. Caracteristici:

- permite apeluri statice şi dinamice
- include un IR scris în Java
- include OSAgent, un serviciu de nume tolerant la defecte

VisiBroker va suporta versiuni Java pentru serviciile Naming, Event, Transactions. Suportă, de asemenea, **Caffeine** un mediu de dezvoltare Java peste CORBA/IIOP. Caffeine face CORBA transparentă pentru programatorii Java: obiecte Java pot invoca alte obiecte prin CORBA IIOP ORB, fără a fi necesare descrieri IDL.

11. Aplicaţii CORBA în Internet

11.1. Modelul cu trei straturi (3-Tiers)

Obiectele CORBA sunt ideale pentru aplicaţiile client-server scalabile, structurate pe trei niveluri.

Primul nivel îl reprezintă aspectele **vizuale** ale obiectelor comerciale (vezi figura 7.4.). Unul sau mai multe **obiecte de vizualizare** pot oferi una sau mai multe vederi ale unui obiect comercial. Aceste obiecte vizuale sunt localizate, de regulă, la client.

Al doilea nivel este cel al **obiectelor-server**, care încapsulează datele persistente şi funcţionalitatea aplicaţiei. Obiectele server interacţionează cu clienţii (obiectele vizuale) precum şi cu sursele de date (baze de date SQL, fişiere HTML, Lotus Notes, Monitoare de tranzacţii) care constituie al treilea nivel al aplicaţiilor. Obiectele server oferă un **model coerent şi integrat** al unor surse de date disparate şi ale aplicaţiilor din fundal. Ele ascund faţă de clienţi (obiectele vizuale) toate detaliile de implementare ale procedurilor şi bazelor de date din al treilea nivel.

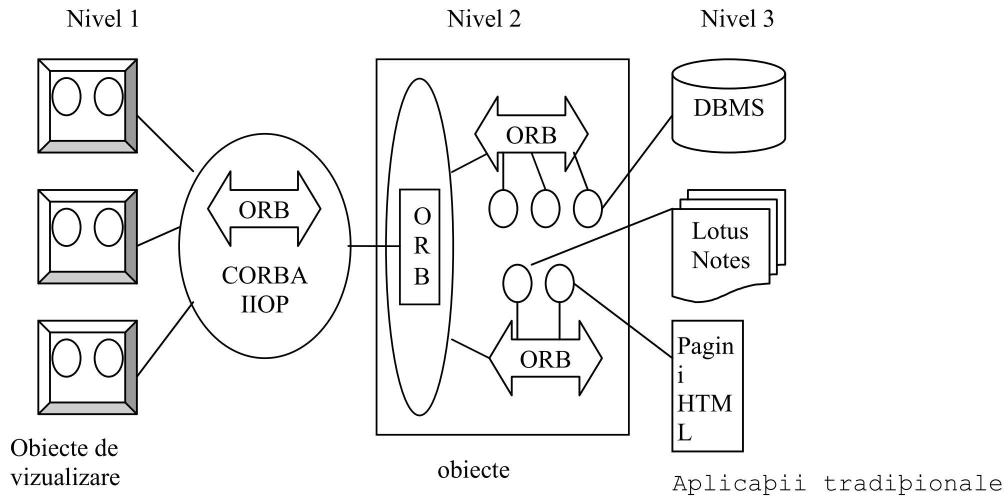

Figura 11.1. Folosirea CORBA în aplicațiile client server

Clienții nu interacționează niciodată direct cu aplicațiile din nivelul trei. Ei comunică cu obiectele server folosind ORB. Serverele pot comunica între ele folosind, de asemenea, ORB. Ele pot, astfel, partaja prelucrările, echilibra încărcarea, etc. Obiectele server comunică cu nivelul trei folosind mijloace tradiționale (mesaje, RPC, etc.). În ce privește Internet-ul, CORBA este o alternativă eficientă la modelul cu trei nivele folosit astăzi, HTTP – Hyper Text Transport Protocol, CGI – Common Gateway Interface (Interfață comună de conversie).

11.2. HTTP / CGI şi CORBA

Reamintim aici, succint, ideea HTTP/CGI. Ea urmăreşte lărgirea funcționalității serverelor Web, adăugând rolului de depozitar de pagini HTML, Hyper Text Markup Language, pe acela de executant al unor prelucrări. O astfel de prelucrare este descrisă printr-un fişier de comenzi (script), căruia i se atribuie un URL. La invocarea unei astfel de "pagini" speciale, atunci când clientul selectează hiper-legătura respectivă, **serverul** lansează în execuție programul (sau fişierul de comenzi). Programul poate inspecta sau actualiza o bază de date sau poate realiza diverse alte prelucrări. Uzual, programul de prelucrare produce şi un fişier de ieşire HTML, care este transmis spre interpretare programului de navigare (clientul). Adăugând la acestea posibilitatea ca un utilizator să transmită informații serverului, prin intermediul clientului (navigatorului) se ajunge la un suport de dezvoltare de aplicații ce beneficiază de toate mecanismele Web pentru interfațarea cu utilizatorii.

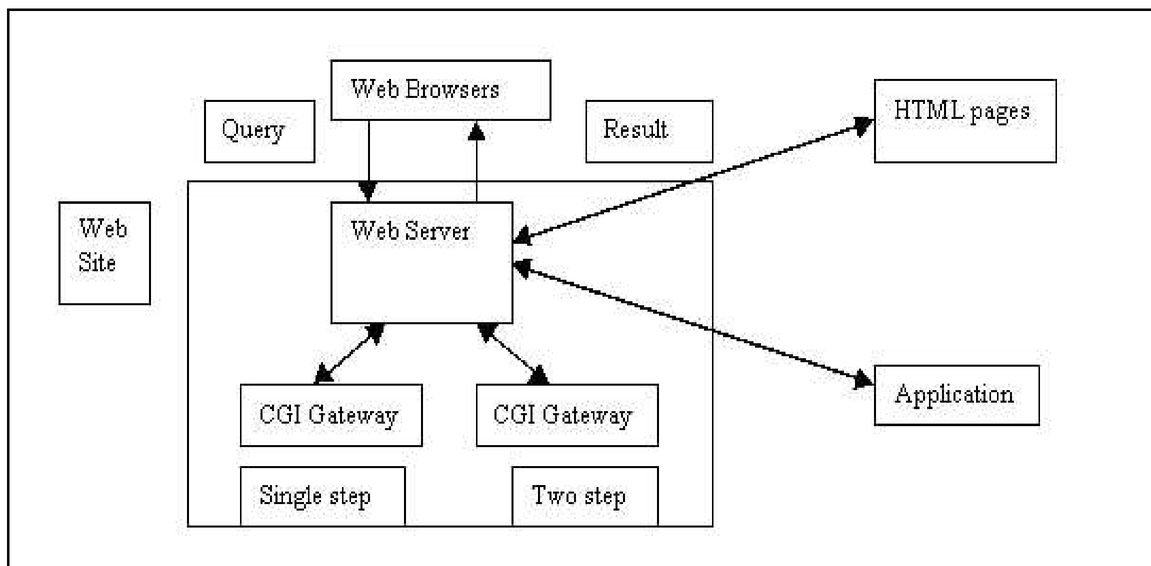

Figura 11.2. Folosirea HTTP/CGI

CGI Gateway

- program rezident în serverul Web
- script (Unix shell, Perl, etc) sau program executabil (C, C++, etc.)
- single step – include şi funcţiile aplicaţiei, care uzual sunt foarte simple (scurte)
- two step – un program de aplicaţie rulează ca un proces **daemon**, iar CGI Gateway joacă rolul de dispecer.

Acest mod de utilizare se bazează pe facilităţi de formă pe care HTML le include şi care permit transmiterea unor informaţii de la browser la server.

Soluţia este lentă şi greoaie.

Java introduce un model nou al interacţiunilor client/server pentru Web. El permite scrierea unor programe, **applets**, care pot fi "încărcate" din server în clienţi (ce sunt Java-compatibili) şi executate de clienţi. Se măreşte astfel interactivitatea şi inteligenţa clienţilor. Java permite crearea unor aplicaţii – client independente de platformă, ce pot fi distribuite prin Internet.

Java este la fel de bun pentru **servere**. Se pot scrie programe pentru servere mobile, utilizabile în combinaţii foarte diverse.

11.3. Java şi CORBA

Permite realizarea unor aplicaţii distribuite portabile. Are RMI (Remote Method Invocation), care permite comunicarea prin Internet între applet-uri: un obiect aflat pe un sistem poate invoca o metodă a unui alt obiect situat într-un alt sistem din reţea. Aplicaţii sunt programe complete, de sine stătătoare, care includ metoda **main()**, ope când un Applet este un mic program executat ca parte a unui browser Java-enabled. Programul conţine metode invocate de browser.

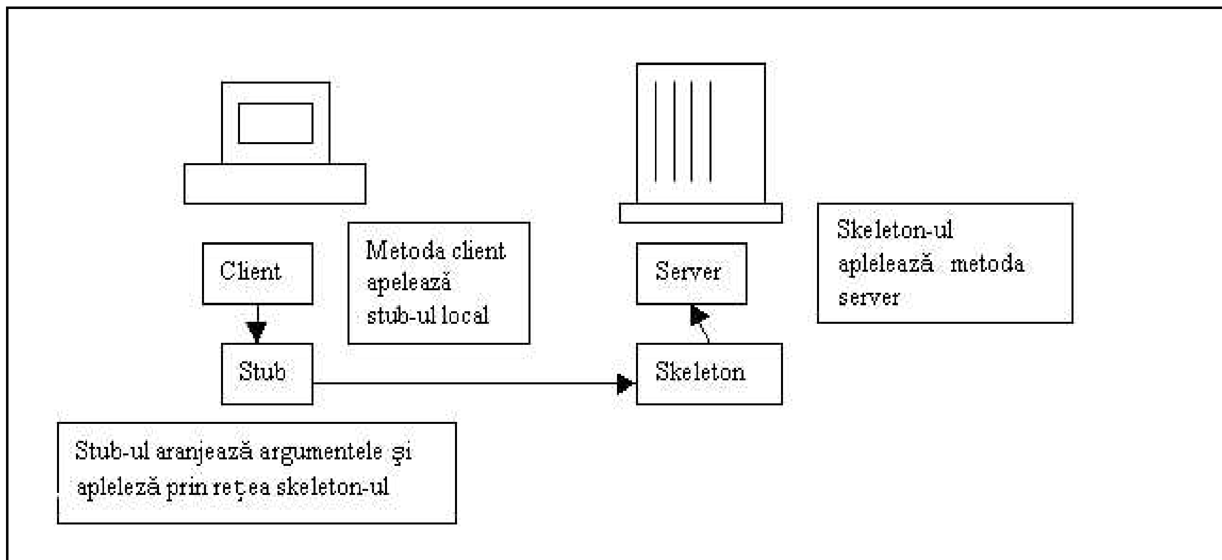

Figura 11.3. Modelul de apel al unui server

Motive pentru utilizarea CORBA:

- Independenţa de limbaj
- Este mai dezvoltată decât RMI (CORBA este înainte de Java).
- Îmbogăţirea infrastructurii Web cu **CORBA** are multe beneficii:

♦ Permite evitarea gâtuirilor CGI la server; clienţii pot invoca direct metode pe un server; pot fi communicate date cu tip nu doar şiruri de octeţi; CORBA păstrează starea între două invocări ale unui client (CGI nu!).

- Permite realizarea unei infrastructuri flexibile de servere, obiectele pot rula pe mai multe servere şi se poate face echilibrarea încărcării.
- CORBA furnizează o infrastructură de obiecte distribuite (serviciile de obiecte şi facilităţile comune).

- Există mai multe **abordări** de a extinde HTTP/CGI cu CORBA – IIOP.
- Realizarea unui convertor CGI->CORBA nu rezolvă gâtuirea CGI deci nu ameliorează performanţele şi nu extinde Java cu o infrastructură distribuită de obiecte.
- Realizarea unor convertoare bidirecţionale HTTP-IIOP şi a unui server CORBA HTTP care serveşte cererile HTTP (aici, CORBA înlocuieşte total HTTP).

Coexistenţa CORBA/IIOP şi HTTP presupune un server CORBA-IIOP alături de serverul HTTP existent; de partea clientului, trebuie să existe un CORBA ORB. Aceasta se poate face fie prin incorporarea unui (Java) ORB într-un program de navigare Web, fie prin încărcarea în client a unui ORB lent (un ORB scris în bytecodes).

Funcţionarea corespunzătoare celei de a treia variente este descrisă în figura 11.4.

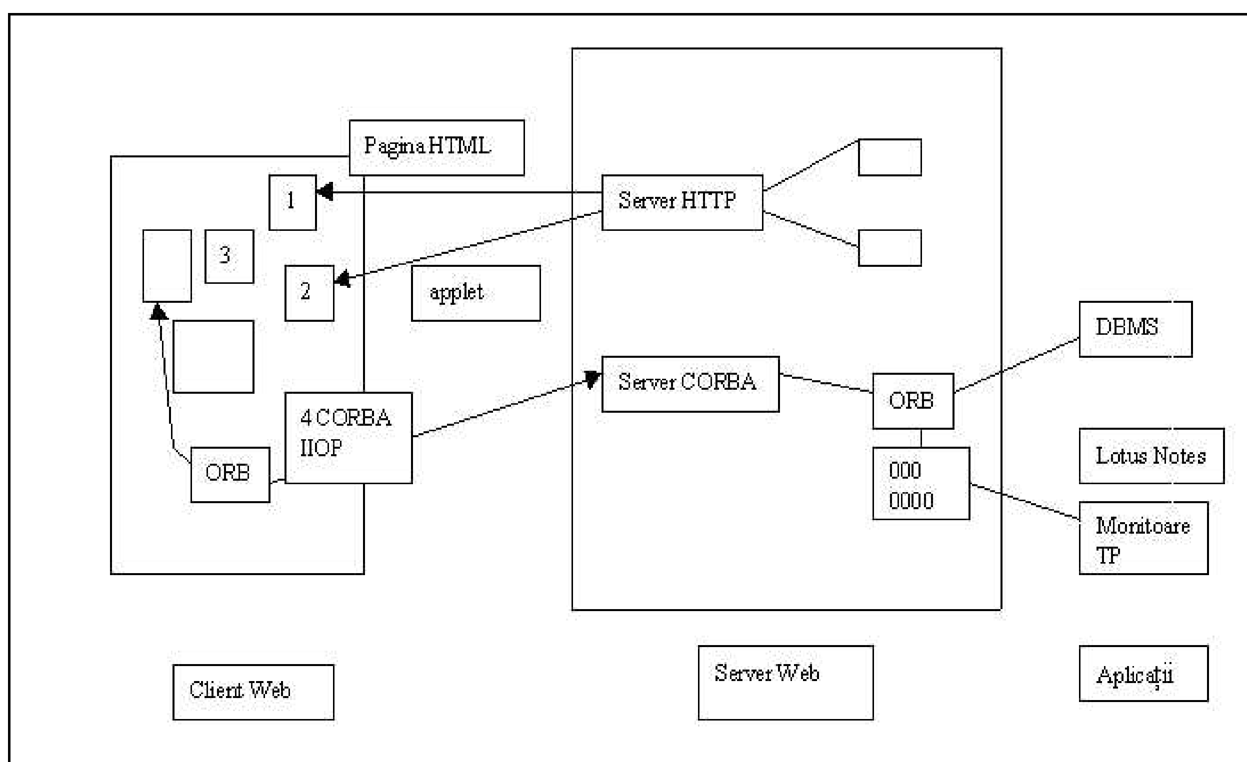

Figura 11.4. Interacţiunea HTTP-CORBA

Interacţiunea ar cuprinde următoarele faze:

- Navigatorul încarcă o pagină HTML, care conţine referinţe la applets Java.
- Navigatorul preia applet-ul de la serverul HTTP.
- Applet-ul este testat şi apoi încărcat în memorie.
- Applet-ul invocă obiecte server CORBA. Sesiunea între ele persistă până când una din părţi decide să se deconecteze.

CORBA reprezintă încă o contribuţie la dezvoltarea Web-ului spre o arhitectură de calcul centrată pe reţea. O astfel de abordare este îmbrăţişată de multe firme. Pe această linie se înscrie **NCA – Network Computing Architecture**, model propus de Oracle pentru medii de calcul distribuite bazate pe Web.

12. Comparație cu alte modele

Dintre produsele de middleware utilizate azi, câteva atrag atenția prin modelul de arhitectură distribuită pe care se bazează:

- CORBA - Common Object Request Broker Architecture al OMG (Object Management Group)
- DCE - Distributed Computing Environment al OSF (Open Software Foundation)
- DCOM - Distributed Component Object Model al Microsoft
- Java RMI al JavaSoft.

Relația între diferitele arhitecturi distribuite este ilustrată în figura 2. Ea arată că aceste modele aflate în competiție se completează reciproc şi se integrează la diferite niveluri.

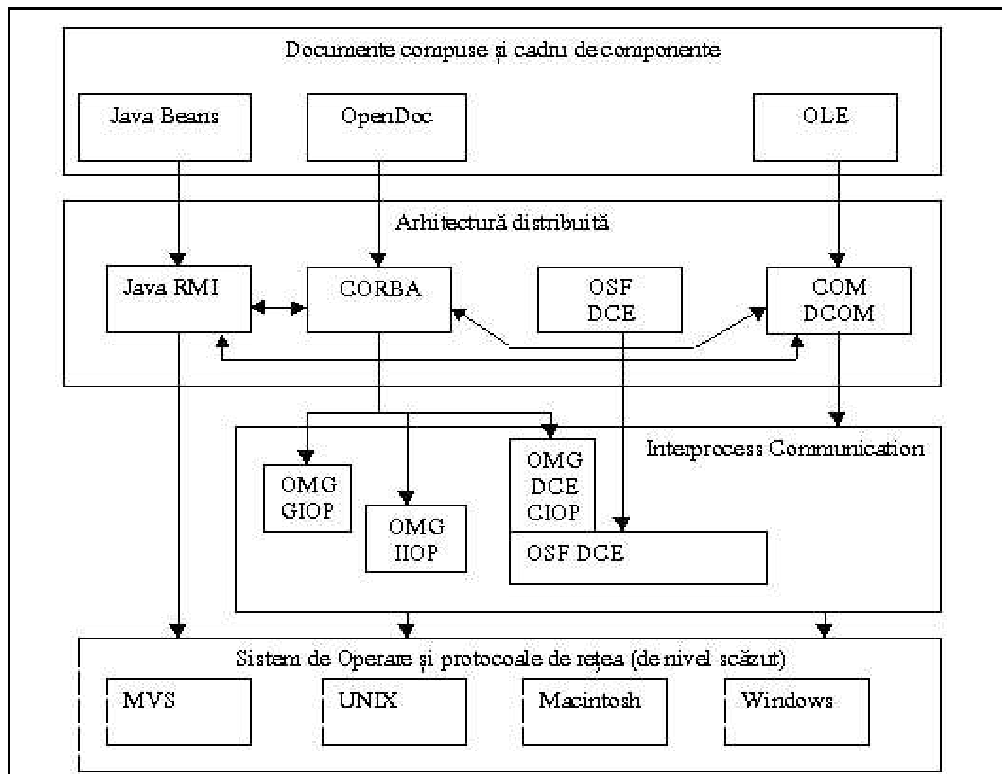

Figura. 12.1. Relația între diferite arhitecturi distribuite

DCE (**Distributed Computing Environment**) este o propunere a **OSF** (**Open Software Foundation**) menită să creeze un mediu-suport pentru aplicații distribuite eterogene. Serviciilor furnizate de rețea, DCE le adaugă facilități pentru thread-uri, apeluri de procedură la distanță, servicii de directoare, de timp, de securitate şi de fişiere. Recunoscând importanța DCE, OMG a introdus un standard de gateway CORBA-DCE. CORBA poate deja opera "peste" DCE, putând folosi facilitățile RPC prin protocolul DCE CIOP.

DCOM (**Distributed Component Object Model**) este modelul propus de **Microsoft** pentru dezvoltarea programelor distribuite orientate pe obiecte. El are multe similarități cu CORBA, dar a fost conceput special pentru platforme Windows. DCOM se referă la:

- Un sistem simplu de transmitere a mesajelor, **DDE** (**Dynamic Data Exchange**);
- Un model de document compus, **OLE** (**Object Linking and Embedding**), cu servicii pentru realizarea de legături între documente compuse sau pentru gestiunea documentelor compuse;
- Un model de comunicare între obiecte, **COM** (**Component Object Model**).

Pentru aplicații Web Microsoft propune **ActiveX**, un set de tehnologii adaptate particularităților Web, incluzând ActiveX Componentns, ActiveX Scripting şi ActiveX documents. Adaptările vizează obținerea unor componente optimizate ca dimensiune şi viteză, ceea ce uşurează încărcarea lor prin

rețea pe mașina clientului. Astfel, componentele ActiveX se conformează modelului COM și abordării OLE.

Java este limbajul care a revoluționat aplicațiile Web. Abordarea standard originală prevedea posibilitatea ca obiecte Java rezidente pe un server să fie transferate la un client pentru execuție. Limbajul nu prevedea servicii distribuite. Obiectele unei aplicații trebuia să rezide într-un singur server, iar obiectele erau "perisabile", pierzându-și informația de stare pe perioada de inactivitate. **Java RMI** rezolvă aceste probleme, fiind începutul dezvoltării unei arhitecturi de obiecte distribuite. Prin el, se pot crea obiecte Java ale căror metode pot fi invocate din alte mașini virtuale. Ca alternativă de evoluție, Java va trebui să integreze capabilitățile distribuite din CORBA sau să-și dezvolte propriile sale capabilități. Prima variantă a și fost abordată: produsul *Caffeine* (produs de Netscape și Visigenic) permite descrierea obiectelor distribuite direct în Java, oferind un suport transparent al RMI peste IIOP. Caffeine generează automat stub-urile și skeleton-urile IIOP. Poate genera chiar și descrierile IDL din bytecode-uri Java. Java are două avantaje importante: este ușor de folosit și codul poate fi încărcat prin rețea și executat. Un client poate localiza un serviciu de interes pentru el, poate încărca interfața corespunzătoare și poate interacționa apoi cu serviciul. Această manieră de lucru este tipică limbajelor de programare a Web-ului.

Lucrarea de față analizează comparativ aceste soluții, principalele rezultate fiind prezentate în tabelele următoare. Ca element de referință a fost ales modelul CORBA, elaborat de OMG, care satisface cerințele unui mare număr de dezvoltatori de aplicații distribuite.

12.1. Comparație CORBA - DCE

CORBA și DCE suportă ambele dezvoltarea aplicațiilor distribuite. Ele au fost realizate de consorții promotoare de standarde, OMG (Object Management Group pentru CORBA), respectiv OSF (Open Software Foundation pentru DCE), au fost specificate în aceeași perioadă (1991-1992) și au cunoscut materializarea în implementări comerciale în 1993.

Caracteristică	CORBA	DCE
Suport pentru	Dezvoltarea aplicațiilor client-server eterogene	Dezvoltarea aplicațiilor client-server eterogene
Servicii comune	Cataloage, timp, threads, securitate, evenimente, gestiune rețea	Cataloage, timp, threads, securitate, evenimente, gestiune rețea
Servicii diferite	Persistență, query, trader, tranzacții, gestiunea informațiilor și a sistemelor, servicii în finanțe, simulare distribuită, CIM	Sistem de fișiere distribuit
Model de programare	Obiect, cu suport pentru încapsulare, abstractizare, polimorfism, moștenire	Procedural; modelul obiect nu este direct suportat
Interfață	Invocare statică și dinamică	Invocare statică
Extensibilitate	Da	Nu
Standardizare	CORBA 1.0 în 1991	DCE 1.0 în 1992
Implementări disponibile în	1993	1993

Diferențele notabile provin din stilurile de programare adoptate: în timp ce CORBA se bazează pe un model de obiecte, DCE are la bază un model procedural, în care apelurile de proceduri la distanță (RPC - Remote Procedure Call) constituie "piesa" centrală.

DCE are un scop mai restrâns, fiind concentrat pe un set redus de servicii. CORBA vizează un set de servicii mai amplu. Ca urmare, implementarea DCE a atins un nivel de maturitate ridicat, în timp ce unele servicii CORBA sunt încă în curs de dezvoltare.

12.2. Comparație CORBA - DCOM

Ca și CORBA, DCOM separă interfața unui obiect de implementarea sa. Limbajul de definire a interfețelor adoptat de Microsoft diferă de CORBA IDL. În timp ce în CORBA moștenirea joacă un rol

esențial, fiecare obiect având o interfață care poate deriva din altele, în DCOM *agregarea* este mai des folosită. Un obiect poate avea mai multe interfețe independente, fără relația de moştenire între ele.

Caracteristică	CORBA	DCOM
Concepție	Ca schemă de interoperare eficientă a aplicațiilor scrise în diferite limbaje pentru platfome diferite (eterogene)	Schemă de integrare a documentelor compuse şi extinsă la prelucrare distribuită
Platforme	MVS, UNIX (diferite versiuni), Windows (toate versiunile), Macintosh	Bine integrat într-o singură platformă, Windows NT; în perspectivă extindere la toate versiunile de Windows, Macintosh, UNIX, MVS
Servicii	Orientat pe anumite categorii de servicii bine definite: persistență, query, trader, tranzacții, gestiunea informațiilor şi a sistemelor, servicii în finanţe, simulare distribuită, CIM	Poate folosi orice servicii oferite de Windows (ActiveX)
Limbaje suportate	C, C++, Smalltalk, Ada95, Java, COBOL	C, C++; în curs - Java, Visual Basic, Ada
Model de bază	Obiect; cu accent pentru persistența şi identitatea obiectelor	Obiect
Interfețe	Separare interfață de implementare Depozit de interfețe	Similar, Type Library
Independență de limbaj	Da	Da
Transparență la localizarea obiectelor	Da	Da
Model de document compus	OpenDoc	OLE
Uşurința de folosire	Mai greu de folosit - util pentru probleme complexe (nu este nativ pe un mediu particular şi cere acomodarea programatorului în fiecare mediu)	Uşor de folosit pentru mediul Windows

Un obiect DCOM nu este un obiect în adevăratul sens al cuvântului. Interfețele DCOM nu pot fi instanțiate şi nu au stări. O interfață DCOM este un grup de funcții, clientului dându-i-se un pointer pentru a avea acces la aceste funcții (mai precis un pointer la un tablou de pointeri la funcții, cunoscut sub numele de *vtable* - virtual table). Aceasta justifică eticheta de standard de "interconectare binară" asociată cu DCOM (respectiv OLE). Deoarece acest pointer nu este legat de vreo informație de stare, un client nu se poate reconecta la o aceeaşi instanță de obiect, la un moment ulterior. Deci, DCOM nu are noțiunea de identitate a obiectului, în timp ce aceasta este foarte bine structurată în CORBA.

Ca şi CORBA, DCOM oferă interfețe statice şi dinamice pentru invocările metodelor, ca şi depozite de interfețe (Type Library). Clienții pot inspecta aceste depozite pentur a descoperi interfețele unui obiect şi parametrii necesari pentru o invocare particulară.

IDL joacă un rol imporant în CORBA, atât pentru a defini sistemul de tipuri, cât şi pentru a preciza modul de transmitere a datelor prin rețea. Similar, ODL (Object Definition Language) din OLE defineşte sistemul de tipuri, iar MIDL specifică modul în care parametrii şi identificatorii operațiilor sunt specificați într-un apel către un obiect OLE. Compilatorul NT 4.0 MIDL extinde IDL pentru a suporta construcțiile ODL, reunind astfel cele două limbaje.

12.3. Comparație CORBA - Java RMI

Relația dintre **Java şi CORBA** este mai mult de complementaritate decât de concurență.

Caracteristică	CORBA	Java RMI
Model	Obiect	Obiect
Creare de obiecte la distanță	Da	Da
Facilități de broker	Da	Da
Limbaje suportate	C, C++, Smalltalk, Ada95, Java, COBOL	Java
Obiecte autodescrise	da	nu
Depozite de obiecte	da	nu
Invocări dinamice	da	nu
Servicii de securitate, tranzacții etc.	da	nu

Java este un excelent limbaj pentru a descrie obiecte CORBA. Apoi, Java complementează serviciul de componente din CORBA, bazat pe OpenDoc. În timp ce CORBA defineşte containere vizuale pentru componente, şi containere de memorare mobile, Java poate furniza "boabele" care evoluează în aceste containere. Mai mult, facilitățile de cod mobil din Java permit partiționarea unei aplicații între client şi server, la momentul execuției. Java simplifică distribuția codului în sistemem CORBA mari, printr-o gestiune centralizată a cosului pe server şi difuzarea codului la clienți, când şi unde este necesar.
În compensație CORBA aduce trei beneficii imediate aplicațiilor Web:

- Permite evitarea gâtuirilor CGI prin invocarea directă, de către client, a metodelor serverelor;
- Facilitează o infrastructură scalabilă de servere (obiectele server pot comunica folosind CORBA ORB)
- Extinde Java cu o infrastructură de obiecte distribuite (posibilitatea de comunicație între spații diferite de adrese).

Avantajele obținute de Java prin conlucrarea cu CORBA sunt următoarele:

- CORBA evită gâtuirea produsă de CGI pe server, deoarece clienții invocă direct metode de pe server. Se pot transfera astfel şi parametri care nu sunt şiruri de caractere, se poate memora starea serverului între două apeluri.
- CORBA asigură o infrastructură scalabilă server-server, putându-se folosi colecții de obiecte server identice care să echilibreze încărcarea.
- CORBA extinde Java cu o infrastructură de obiecte distribuite, astfel încât un applet este în stare să comunice cu orice obiect, indiferent de limbajul în care este scris şi de localizarea în rețea.

Sintetizând, diferențele dintre abordarea HTTP şi CORBA sunt prezentate în următorul tabel:

Caracteristică	Java – CORBA	Java - CGI
Păstrarea stării între invocări	DA	NU
IDL, Interface Repository	DA	NU
Suport metadate	DA	NU
Invocare dinamică	DA	NU
Tranzacții	DA	NU
Securitate	DA	DA
Servicii bazate pe obiecte	DA	NU
Callbacks	DA	NU
Infrastructură server-server	DA	NU
Scalabilitate	DA	NU
Metode IDL	DA	NU

CORBA nu este singura alternativă, competitorii săi fiind DCOM/ActiveX, WebObjects şi RMI. Însă alăturarea dintre Java şi CORBA aduce o serie de avantaje şi pentru CORBA, Java intrând în funcțiune acolo unde CORBA se termină:

- Java permite CORBA deplasări de "inteligență", prin folosirea codului mobil atât clienții cât şi serverele putând să-şi îmbogățească cunoştințele.

- Java completează serviciile CORBA referitoare la controlul ciclului de viață al obiectelor.
- Java simplifică distribuirea codului în sistemele CORBA de dimensiuni mari.
- Java complementează facilitățile CORBA de lucru cu agenți mobili.
- Java este limbajul cvasi-ideal pentru scrierea obiectelor CORBA.

DCOM (Distributed Component Object Model) este principalul rival al alianței Java/CORBA, considerat actualmente un standard "de facto", datorită răspândirii sale. Este fundamentul viziunii Microsoft asupra Internet-ului.

DCOM, la fel ca şi CORBA, separă interfața de implementare cerând ca toate interfețele să fie descrise folosind un IDL. Microsoft IDL se bazează pe DCE şi nu este compatibil cu CORBA (aşa cum era de aşteptat☺). În timp ce CORBA se bazează pe modelul clasic de obiecte, DCOM nu face acest lucru. O componentă DCOM nu suportă moştenire multiplă, însă poate implementa mai multe interfețe, astfel încât reutilizarea codului nu se obține prin moştenire, ci prin agregare.

Sintetizând, principalele aspecte legate de comparația Java/CORBA – DCOM sunt prezentate în tabelul următor:

Caracteristică	Java/CORBA	DCOM
ORB scris în java	DA	NU
Platforme suportate	Toate	Windows
Transfer de parametri	In, out, in/out	In, out, in/out
Configurare	Uşoară	Grea
Descărcare dinamică a stub-urilor	NU	NU
Descărcare dinamică a claselor	NU	NU
Transfer obiecte prin valoare	DA	NU
Interfața - IDL	DA	DA
Aflare dinamică a informațiilor	DA	DA
Invocare dinamică	DA	DA
Performanțe	F. rapid	Rapid
Securitate	DA	DA, pe NT
Serviciu de nume	DA	NU
Recunoaşte URL	DA	NU
Referințe persistente la obiecte	DA	NU
Invocații multilimbaj	DA	DA
Scalabilitate	DA	NU
Protocol ORB standard	DA	Posibil

Deşi CORBA a luat un start promițător, DCOM este o amenințare serioasă la supremația Web-ului.

Făcând acum sinteza tehnologiilor Internet existente, putem vedea că, per ansamblu, CORBA deține cele mai bune performanțe. Acest lucru este ilustrat în tabelul următor, unde se foloseşte un sistem de notare asemănător celui de apreciere a filmelor: cinci stele maxim, o stea minim.

Caracteristică	CORBA	DCOM	RMI	HTTP/ CGI	Sockets
Nivel de abstractizare	*****	*****	*****	***	**
Integrare Java	*****	****	*****	***	***
Sisteme de operare suportate	*****	***	*****	*****	*****
Implemenatare Java	*****	**	*****	*****	*****
Parametri cu tip	*****	*****	*****	**	**
Uşurință de configurare	****	*	****	****	****
Invocare de metode distribuite	*****	****	****	*	*
Salvare stare într invocări	*****	****	****	*	***
Invocare dinamică	*****	*****	**	*	*

Caracteristică	CORBA	DCOM	RMI	HTTP/ CGI	Sockets
Performanțe	*****	****	****	*	*****
Securitate	*****	*****	****	****	****
Tranzacții	*****	****	*	*	*
Obiecte persistente	*****	**	*	*	*
Recunoaştere UTL	****	**	***	*****	****
Invocare multilimbaj	*****	****	*	****	*****
Scalabilitate	*****	**	**	***	*****
Standard deschis	*****	***	***	*****	*****

13. Concluzii

În lucrare sunt discutate motivele pentru utilizarea tehnologiei **CORBA:**

- Independența de limbaj şi de platformă
- Tehnologia este mai dezvoltată decât RMI (CORBA este propusă înainte de Java).
- Îmbogățirea infrastructurii Web cu **CORBA** are mai multe avantaje:
- Permite evitarea gâtuirilor CGI la server; clienții pot invoca direct metode pe un server; pot fi communicate date cu tip nu doar şiruri de octeți; CORBA păstrează starea între două invocări ale unui client (CGI nu!).
- Permite realizarea unei infrastructuri flexibile de servere, obiectele pot rula pe mai multe servere şi se poate face echilibrarea încărcării.
- CORBA furnizează o infrastructură de obiecte distribuite (serviciile de obiecte şi facilitățile comune).

Bibliografie selectivă

1. S.Baker, CORBA Distributed Objects Using Orbix, Addison Wesley 1997
2. T.J.Mowbray, W.A.Ruh, Inside CORBA, Distributed Object Standards and Applications, Addison Wesley 1997
3. D.E. Comer, D.L. Stevens *Internetworking With TCP/IP* Prentice Hall, 1993
4. S. Beaker, V.Cahill, P.Nixon, *Bridging Boundaries: CORBA in Perspective*, în IEEE Internet Computing Vol.1, No.5, Sept/Oct 1997
5. E.Evans, D.Rogers, *Using Java applets and CORBA for Multi-User Distributed Applications*, în IEEE Internet Computing Vol.1, No.3, May/June 1997
6. C. McFall, An Object Infrastructure for Internet Middleware; IBM on Component Broker, în IEEE Internet Computing Vol.2, No.2, March/April 1998
7. R. Ben Natan *Objects on the Web*, McGraw Hill 1997
8. A. Umar, Object Oriented Client/Server Internet Environments, Prentice Hall 1997
9. www.microsoft.com, DCOM Architecture
10. www.sun.com, Java RMI Specification
11. J. Boutell , *CGI Programming in C and Perl*, Addison Wesley, 1997