Příklad použití distribuovaných objektů

Vilém Vychodil

21. července 2001

Abstrakt

Tento text by měl sloužit jako dokumentace k vytvořeným ukázkovým příkladům. Cílem příkladů bylo demonstrovat použití standardu CORBA pro komunikaci objektů v heterogenním distribuovaném prostředí. Pro implementaci byly zvoleny OpenSource implementace standardu CORBA — MICO a JacORB. Zdrojové kódy jsou napsány v C++ (klientská a serverová část) a v Javě (klientská část).

1. Úvod do problematiky

Většina moderních programovacích jazyků využívá ke strukturování výpočtu *objektové paradigma*. Přirozeným rozšířením výpočetního procesu je jeho distribuce na několik nezávislých uzlů, které mohou fyzicky existovat na jednom nebo i více počítačích. Jelikož se objektové paradigma ukazuje výhodné pro vytváření velkých softwarových děl, je přirozené jej používat i při vytváření distribuovaných aplikací. Použitím objektového stylu programování v distribuovaném prostředí vzniká systém nezávislých objektů, které spolu komunikují přes síťové prostředí.

V prostředí rozlehlé počítačové sítě se lze setkat s počítači různých platforem ať už hardwarových, nebo softwarových. Velké rozdíly jsou i v jednotlivých objektově orientovaných jazycích, v jejich pohledu na samotný pojem *objekt*, na *stupeň ochrany dat*, *perzistentní uložení dat* a podobně. Pro zajištění komunikace mezi objekty v takovém heterogenním prostředí je potřeba jasně definovat standard a mít k disposici softwarový middleware, který jej implementuje.

Jednou z komunikačních platforem je CORBA — *Common Object Request Broker Architecture*, specifikovaná OMG. Jako příklad komunikace distribuovaných objektů byl zvolen jednoduchý „bankovní systém“. Příklady jsou implementovány ve dvou jazycích — C++ a Java. Kód v C++ využívá volně šiřitelnou implementaci standardu CORBA — MICO. Kód v Javě využívá volně šiřitelnou implementaci JacORB. Serverová část je napsána pouze v C++, klientská část je napsána v obou jazycích. Jelikož Java vyniká především svou platformovou přenositelností, lze klientskou část používat na systémech UNIX, Windows, OS/2, MacOS a dalších.

2. Popis implementace

Cílem je naimplementovat funkční „bankovní systém“ sestávající z třídy banka, která poskytuje účty. Na jednotlivé účty je možno připisovat hotovost, vybírat hotovost a informovat se o jejich stavu. Během života aplikace bude existovat jeden objekt *banka*, na který se budou obracet klienti s požadavky na vytvoření nového účtu.

V tomto ukázkovém případě je jasně vidět předěl mezi jednotlivými prvky systému. Na jedné straně jsou to objekty *banka* a *účty*, na straně druhé jsou to klienti. Klienti komunikují s bankou přes nějaké existující rozhraní, které musí být přesně dáno. Rozhraní musí být definováno dostatečně nezávisle, protože klienti i bankovní systém mohou být implementováni na zcela jiných platformách. Situace je analogická obsluze skutečného zákazníka v bance. Jeden zákazník může jít k přepážce do více bank, ale jejich vnitřní organisace je mu obvykle utajena. Stejně tak bance je v zásadě jedno, jakou řečí mluví zákazník, pokud za něj mluví peníze.

2.1. Definice rozhraní

Jelikož CORBA podporuje řadu programovacích jazyků, musí být nejprve odděleno *rozhraní* od *implementace*. Implementace je vytvořena pomocí zvoleného programovacího jazyka. Rozhraní — *interface* je definován pomocí speciálního jazyka IDL — *Interface Definition Language*. Zjednodušeně řečeno, CORBA IDL je velmi podobný jazyku C++, je redukovaný pouze na *deklarace objektů* a *typů*.

Rozhraní bankovního systému vypadá následovně.

```
// Account Interface
interface Account {
    void deposit (in unsigned long amount);
    void withdraw (in unsigned long amount);
    long balance ();
};

// Bank Interface
interface Bank {
    Account create ();
};
```

Jednotlivá rozhraní jsou definována v blocích uvozených klíčovým slovem **`interface`**. Při specifikaci formálních argumentů bylo uvedeno pomocné klíčové slova **`in`**. Pomocí něj se dává na vědomí, že daný argument je pouze *vstupní* — při volání metody je předána hodnota vstupního argumentu.

Dalším krokem spuštění *IDL kompilátoru*, který na základě předloženého rozhraní vygeneruje kód ve zvoleném cílovém jazyku, například C++. Použijeme-li IDL kompilátor, který je součástí implementace MICO, stačí v příkazovém interpretu spustit

```
$ idl --poa --no-boa account.idl
```

IDL kompilátor vygeneruje dva soubory, `account.h` a `account.cc`. Přepínače uvedené při spuštění IDL kompilátoru způsobí vygenerování skeletonu pro POA — *Portable Object Adapter*. Soubory `account.h` a `account.cc` jsou déle využíván pro implementaci serverové i klientské části aplikace. Ke každému rozhraní uvedenému v IDL obsahují vygenerované soubory implementaci *tří základních tříd*.

Uvažujme rozhraní `Account`. První z tříd je základní třída `Account` obsahující všechny definice, které patří k rozhraní. Pro každou metodu obsaženou v rozhraní, tato třída definuje čistě virtuální metodu stejného jména. Deklarace třídy `Account` bude vypadat následovně,

```
// Basic Account Class
class Account {
...
public:
    virtual void deposit (CORBA:: ULong amount) = 0;
    virtual void withdraw (CORBA:: ULong amount) = 0;
    virtual CORBA:: Long balance () = 0;
...
};
```

Třída `Account` nemůže být instanciována, ale je společným předkem dalších tříd. Třída `POA_Account` je obohacuje třídu `Account` o *dispatcher* a je základní třídou pro implementaci metod v objektu. Třída `Account_stub` je potomkem `Account` a narozdíl od `POA_Account` implementuje právě všechny virtuální metody. Implementace je generovaná pomocí IDL kompilátoru, jednotlivé metodu třídy `Account_stub` realizují marshalling při volání metod objektu.

2.2. Implementace tříd

Bezprostředně po vygenerování kódu z IDL souboru je programátor postaven před úkol realisovat buďto klientskou část, nebo serverovou. Klient je ta část distribuovaného programu, která si vyžádá instanci objektu a pracuje s ním. Server je opačná strana, poskytuje objekty a realisuje jejich implementaci. Nyní se budeme zabývat právě tímto problémem, jak provést implementaci rozhraní definovaných v IDL.

Pro každé rozhraní je potřeba vytvořit samostatnou třídu, která bude odvozená z třídy `POA_`*jméno*. Nejprve uveďme implementaci třídy `Account`, která je velmi přímočará.

```
// Account Implementation
class Account_impl: virtual public POA_Account {
    // account balance
    CORBA:: Long bal;
public:
    // constructor and public methods
    Account_impl ();
    void deposit (CORBA:: ULong);
    void withdraw (CORBA:: ULong);
    CORBA:: Long balance (void);
};

Account_impl:: Account_impl () { bal = 0; }
void Account_impl:: deposit (CORBA:: ULong amount) { bal += amount; }
void Account_impl:: withdraw (CORBA:: ULong amount) { bal -= amount; }
CORBA:: Long Account_impl:: balance (void) { return bal; }
```

Třída obsahuje jeden privátní atribut, stav účtu. Jelikož se jedná o privátní člen třídy, nebyl v rozhraní třídy definován. To je opět analogické situaci při skutečném placení v bance. Zákazníkovi je principiálně jedno, jestli si banka informace ukládá do počítače nebo na papír. Implementace třídy `Bank` je o něco komplikovanější, protože metoda `create()` dle rozhraní vrací referenci na objekt `Account`.

```
// Bank Implementation
class Bank_impl: virtual public POA_Bank {
public:
    // create a new account
    Account_ptr create (void);
};

Account_ptr Bank_impl:: create (void)
{
    Account_impl *ai;      // new account implementation
    Account_ptr a_ref;     // account reference

    // create new account
    ai = new Account_impl;
    a_ref = ai-> _this ();

    // return the reference
    return a_ref;
}
```

Úkolem metody `create()` je vrátit referenci na nově vytvořený objekt. Ve svém těle nejprve vytvoří novou instanci třídy `Account_impl`. Dále vytvoří odkaz pomocí volání metody `_this()`, kterou implementuje třída `POA_Account`. Je nutné si uvědomit, že pracujeme s distribuovanými objekty. Korektní referenci nelze získat použitím operátoru `&`.

Po implementaci obou tříd ještě musíme vyřešit několik problémů. Jednak je na straně serveru potřeba spustit ORB tak, aby byl schopen obsloužit příchozí požadavky ze stran klientů. S tím souvisí další problém, jak objekt v distribuovaném prostředí lokalisovat.

2.3. Identifikace objektů

Na straně serveru je nutné zajistit aktivaci objektu `Bank` a dát jej věřejně k disposici klientům. Při komunikaci serveru a klienta se na obou stranách využívá služeb ORBu. ORB je representován jedním objektem třídy `CORBA:: ORB_var`, který je na počátku práce inicialisován. Před aktivací objektu je na straně serveru potřeba inicialisovat ORB a POA. Začátek hlavní funkce serveru bude vypadat následovně.

```
// main function
int main (int argc, char **argv)
{
    CORBA:: ORB_var orb;                        // ORB
    CORBA:: Object_var poaobj;                  // POA object
    PortableServer:: POA_var poa;               // POA server
    PortableServer:: POAManager_var mgr;        // POA manager

    // ORB initialization
    orb = CORBA:: ORB_init (argc, argv);

    // POA initialization
    poaobj = orb-> resolve_initial_references ("RootPOA");
    poa = PortableServer:: POA:: _narrow (poaobj);
    mgr = poa-> the_POAManager ();
```

Kořenový POA server je získán voláním `resolve_initial_references`, samotný objekt je záskán pomocí *downcastu* na následujícím řádku. Downcast je přetypování na objekt, jehož třída ležící v hierarchii níže. S kořenovým POA serverem je spojen POA Manager, ten může kontrolovat oběcně více než jeden POA server. POA Manageru je odpovědný za prvotní manipulací s příchozími požadavky. Příchozí požadavky lze buďto *okamžitě obsluhovat*, *řadit do fronty* nebo *zahazovat*. Implicitně jsou požadavky po jeho startu řazeny do fronty.

Funkce `main ()` pokračuje kódem, který vytvoří banku a provede její aktivaci.

```
    Bank_impl *bank;                            // bank implementation
    PortableServer:: ObjectId_var oid;          // object identifier
    CORBA:: Object_var ref;                     // reference
    CORBA:: String_var ior;                     // object IOR string

    // create and activate the bank
    bank = new Bank_impl;
    oid = poa-> activate_object (bank);

    // create a string reference
    ref = poa-> id_to_reference (oid. in ());
    ior = orb-> object_to_string (ref. in ());

    // start the indefinite loop
    mgr-> activate ();
    orb-> run ();
}
```

Z předchozího kódu je nejzajímavější vytvoření reference ve tvaru řetězce. Distribuované objekty jsou lokalisovány ORBem pomocí IOR řetězců. V IOR řetězci jsou zakódovány informace o lokaci objektu, řetězec je dlouhý několik set znaků a jeho unicita v Internetu je zajištěna například tím, že je vypočítáván i z MAC adresy síťové karty. Implementace MICO disponuje programem `iordump`, který umožňuje zobrazit informace z řetězce v čitelné podobě.

2.4. Mechanismus předání IOR řetězce

Předchozí kód by byl v uvedené podobě k ničemu. K objektu byla sice vytvořena jeho řetězcová reference, ta ale nebyla nijak předána nebo zpřístupněna. IOR řetězec může být klientům předán různým způsobem, elektronickou poštou, pomocí jmenných služeb nebo například pomocí sdíleného souborového systému.

V ukázkové implementaci byl zvolen přenos přes TCP/IP pomocí socketu. *Socket* je mechanismus, kterým je možno zprostředkovat lokální či vzdálenou komunikaci dvou uzlů, která má charakter *klient/server*. Toto žešení znamená z implementačního hlediska asi tolik, že serverová část programu se rozdělí na dvě vlákna. Jedno z nich obsluhuje příchozí požadavky na banku, druhé vlákno čeká na případné požadavky na vyhrazeném portu a klientům zasílá IOR string.

2.5. Implementace klienta

Pokud se klient chce spojit s bankou, nejprve si vyžádá IOR string, k tomu použije doménové jméno počítače a předem smluvený port. Po té, co získá řetězec, může pracovat s bankou. Popis pomocí socketu nebude dále rozebírán, čtenář jej nalezne ve zdrojových kódech programu.

Implementace klienta je výrazně jednodušší.

```
int main (int argc, char **argv)
{
    CORBA:: ORB_var orb;          // ORB
    CORBA:: Object_var obj;       // Object
    Bank_var bank;                // Bank
    Account_var account;          // Account
    char *ior;                    // IOR string

    // ORB initialization
    orb = CORBA:: ORB_init (argc, argv);

    // get IOR string
    ior = get_ior_string (...);

    // get bank reference
    obj = orb-> string_to_object (ior);
    bank = Bank:: _narrow (obj);

    // obtain new account
    account = bank-> create ();

    // call methods
    account-> deposit (700);
    account-> withdraw (450);
    cout << "Balance is " << account-> balance () << endl;

    return 0;
}
```

Předchozí kód je schématický. Skutečný program by měl po každé získané referenci testovat, zda-li objekt není nulový. Nulovost opět nesmí být testována klasickým způsobem, ale pomocí metody `CORBA:: is_nil()`, například

```
if (CORBA:: is_nil (bank)) {
    cerr << "The bank is unreachable." << endl;
   _exit (1);
}
```

Odkazy

[1] *JacORB HomePage.*
`http://www.jacorb.org`.

[2] *MICO HomePage.*
`http://www.mico.org`.

[3] Vychodil, V. *Komunikace pomocí Socketu.*
Referat, `http://www.inf.upol.cz/~vychodil`.