CORBA

Ing. Jan Pečiva, peciva@fit.vutbr.cz
Ústav počítačové grafiky a multimédií
Fakulta informačních technologií
Božetěchova 2, 612 00 Brno

CORBA (Common Object Request Broker Architecture) je platformě nezávislá architektura a infrastruktura, kterou aplikace používají k vzájemné spolupráci. Z historického hlediska byla první verze standardu CORBy vypuštěna skupinou OMG (http://www.omg.org) v roce 1991. Jako zajímavost z [8] by se dalo uvést, že příchod CORBy znamenal začátek konce RPC (Remote Procedure Call), který se sice hojně používá v databázích, či NFS (unixový filesystém po síti), ale tak jako objektové programování vytlačilo funkcionální, tak se zdá, že CORBA, představující objektový přístup, vytlačí RPC. Ve skutečnosti se RPC používá už pouze tam, kde má svoje pevné kořeny. Do nových systémů už snad pouze CORBA.

Jednoduše řečeno, CORBA umožňuje vytvářet aplikace spolupracující po síti. Přitom je naprosto jedno, jaké architektury, či operačního systému je na jednotlivých strojích použito. Klíčovým prvkem je přitom oddělení rozhraní od implementace. V CORBě vždy hovoříme o rozhraní a implementaci objektů, přestože vše může být realizováno v neobjektovém jazyce (například C). Standard pro definici rozhraní poskytuje jazyk IDL (Interface Definition Language). Tento jazyk je nezávislý na konkrétním použitém programovacím jazyku, proto i CORBA programy mohou být psány ve většině používaných jazyků (C, C++, Java, Lisp,...).

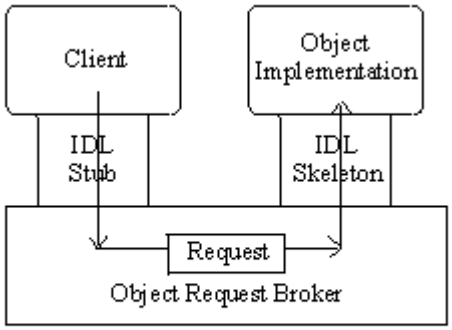


Figure 1: A request passing from client to object implementation

Základní funkce je ukázána na obrázku 1, který pochází z [2]. Z rozhraní, popsaného jazykem IDL, se v době vytváření programu vygeneruje IDL stub a IDL skeleton. Stub je rozhraní objektu, jehož hlavní funkcí je zpakovat parametry předané objektu při volání jeho funkce do streamu (proces zvaný marshaling). Naopak IDL skeleton zase rozbalí tyto parametry a vyvolá na nich danou funkci (unmarshaling). Případná návratová hodnota se zpět přenese k volajícímu objektu.

Srdcem CORBy je ORB (Object Request Broker) [3]. Nejdůležitější úlohou ORB je přenos požadavku od klienta k implementaci objektu a následný přenos odpovědi zpět. Toto není zdaleka triviální úkol. Klient a vlastní implementace objektu totiž mohou sídlit na různých počítačích, různých architekturách a jiných programovacích jazycích, spojených exotickou sítí, kde se nepoužívají k identifikaci IP adresy, a dokonce se volaný objekt nemusí ani nacházet v paměti - může být uvolněn a jeho stav uložen na disk pro úsporu zdrojů.

Klíč k možnosti volat objekt je získat jeho referenci. CORBA pro referenci zavádí pojem IOR (Interoperable Object Reference) [7]. Je to relativně dlouhý klíč, který identifikuje daný objekt. Ten v sobě obsahuje především routovací informace, které slouží k identifikaci počítače, na kterém objekt přebývá, a 128-mi bitový náhodně vygenerovaný klíč, který garantuje jedinečnost klíče s více než dostatečnou jistotou, jak říká [9]. Takže se nestane, že by po restartu počítače, počítač vytvořil nové objekty se stejnou identifikací, které na něm sídlily před restartem. Taková věc by

totiž mohla činit dost velké problémy aplikacím z okolních počítačů.

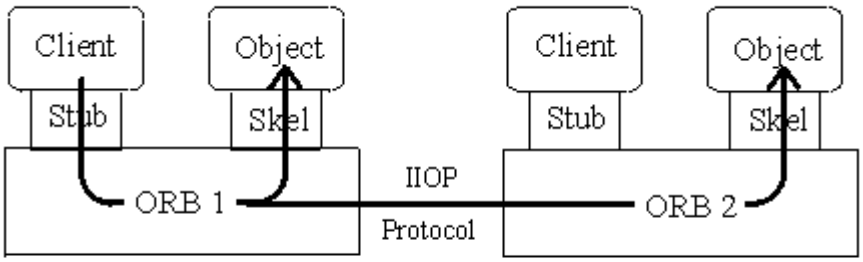


Figure 2: Interoperability uses ORB-to-ORB communication

Na obrázku 2 opět z [2] vidíme, jak probíhá lokální a vzdálené vyvolání funkce objektu (anglický termín: object invocation). Je zavolána funkce objektu. Na klientském rozhraní (Stub) dojde ke sbalení parametrů a ORB (Object Request Broker) dekóduje identifikaci objektu (IOR). Z ní zjistí cílový počítač, na kterém sídlí volaný objekt. Pokud je cílový počítač shodný se zdrojovým, tak se vše provede lokálně, jinak se IIOP (Internet Inter-ORB Protocol) [7] spojí místní ORB s ORB na cílovém počítači a vyvolání objektu se provede tam. Vše transparentně k uživateli, pouze rychlost volání je jiná. IIOP protokol je velmi přesně standardizován, proto zdrojový i cílový ORB mohou pocházet od různých výrobců a mohou běžet na jiných platformách. IIOP protokol specifikuje komunikaci skrze internet. Podobně mohou znikat jiné protokoly, komunikující například IPX protokolem nebo přes sériové porty počítačů. Všechny tyto nové protokoly by však měly být založeny, stejně jako IIOP, na GIOP (General Inter-ORB Protocol), který specifikuje základy celého inter-ORB komunikačního protokolu, avšak bez specifičností, které se vztahují pak k použitým síťovým protokolům.

Nyní můžou navazovat otázky, jak je tedy vlastně ta CORBA rychlá? A dá se použít při vývoji klasických aplikací? Při studiu těchto otázek jsem vyšel především z informací, které se vztahovaly ke knihovně ORBit (http://orbit-resource.sourceforge.net/), což je open-source implementace CORBy (licence LGPL). ORBit se používá v okenním systému GNOME (Unixy a Linux), takže jej s velkou pravděpodobností máme i v našich Linuxových strojích. ORBit se v GNOME používá, dle slov vývojářů GNOME, všude – File Manager, Panel, Window Manager a Bonobo (Bonobo je obdoba OLE z Windows pro GNOME aplikace). Představa byla taková, že aplikace bude schopna využívat služeb jiných, například textový editor si zavolá jinou aplikaci pro vytvoření vloženého objektu grafu z nějakých dat. Nad těmito daty si tato druhá aplikace zavolá aplikaci například kalkulačky (GNOMovská kalkulačka skutečně má CORBA rozhraní). Ta spočítá požadovaná data a nakonec ve výsledku dostane textový editor to, oč žádal. Tento přístup je mnohem čistší než přístup COM, protože, jak uvádí [8], skrz CORBu je pro aplikaci transparentní, zda-li aplikace pracuje sama o sobě nebo zda-li ji využívá jiná aplikace jako CORBA server.

Další otázka je rychlost CORBy. ORBit o sobě tvrdí, že je velmi rychlá knihovna s malými nároky na paměť. Malé paměťové nároky jsou vynuceny její hojnou přítomností v GNOME aplikacích. Rychlost při komunikaci se vzdáleným objektem je vždy nižší o latenci sítě – zde proto můžeme samotnou rychlost CORBy zanedbat. Pokud ale pracujeme lokálně tak, jak to platí v případě GNOME a jeho serverovských aplikací, tak požadujeme co největší rychlost. Mánie po rychlosti šla v tomto případě až do krajnosti, a proto komponenty, které jsou označeny jako bezpečné, je možné dokonce zavádět do adresového prostoru volajícího programu, jak říká [8]. Tím se dramaticky sníží

prodleva, kterou už je pak možno měřit s voláním virtuální funkce. CORBA volání sice vyjde dráže, ale už jen o dobu konverze reference (IOR) na skutečný ukazatel do paměti, což se dá hešovací funkcí udělat relativně velmi rychle. V případě rozdílných adresových prostorů dochází, tuším, k přepnutí úlohy, což může být asi stejně náročné, jako volání funkce jádra OS. Proto je například nevhodné implementovat třídu pro řetězce jako CORBA objekty, či procházet sto prvků seznamu a pro každý volat CORBA funkci. Lépe jedné funkci předat seznam sta prvků a vše zavolat pouze jednou.

Velmi častou otázkou začátečníků v CORBě je: A jak připojím klienta k serveru? Za touto otázkou se skrývá problém anglicky zvaný "bootstrapping" (problém je detailněji diskutován v [10]). V oblasti počítačových síti pošle klient broadcast a server mu odpoví. V CORBě ale žádný broadcast nemáme, zde se používá jiných technik. Například v GNOME se při startu spustí serverovská aplikace, která má CORBA rozhraní a čeká, kdo bude využívat jejich služeb. O něco později spustí uživatel svou aplikaci, která by ráda využívala služeb daného serveru. Avšak k tomu, aby jej mohla využívat, potřebuje znát referenci (IOR) objektu, který serverovská aplikace vypublikovala ve svém CORBA rozhraní. Jednou z možností je zadat IOR na příkazový řádek clientské aplikace, což ale není příliš pohodlné, nebo využít některých možností meziprocesové komunikace a IOR si vyměnit tímto způsobem. GNOME daný problém řeší uložením speciálního souboru do temp adresáře, který obsahuje náš IOR. Klientská aplikace si pak daný soubor přečte a skrz IOR naváže komunikaci se server objektem.

CORBA definuje i poněkud přímější řešení pro navázání komunikace mezi klientem a serverem. K tomu používá Naming Service [12] a Trader Service [13]. Naming Service je známá také jako CosNaming. Umožňuje registrovat každý objekt pod jedinečným jménem. Klienti, se pak mohou díky této službě snadno získat reference (IOR) na objekty, které požadují. Trader Service je poněkud mocnější služba, neboť umožňuje vyhledávat služby podle specifikovaných požadavků, které na službu máme. Například se můžeme dotazovat na tiskárnu s podporou postscriptu, černobílou, na třetím podlaží budovy a s co nejvyšším rozlišením. Trader nám následně odpoví, zda-li objekt splňující tyto požadavky byl zaregistrován a vrátí nám jeho referenci. Nezávislí traders mohou být spojováni do "federací", které umožní vyhledávání v rozsáhlé síti. Naming Service nám naopak umožňuje omezit prohledávaný prostor definováním kontextu, který zabrání kolizi jmen, či navrácení úplně jiných objektů díky shodě jmen. Jako perla by se dalo dodat, že v roce 2000 bylo standardizováno IDL rozhraní objektu Naming Service, takže nyní je možné se dotazovat na objekty na jiných počítačích jiných architektur a jiné implementace CORBy.

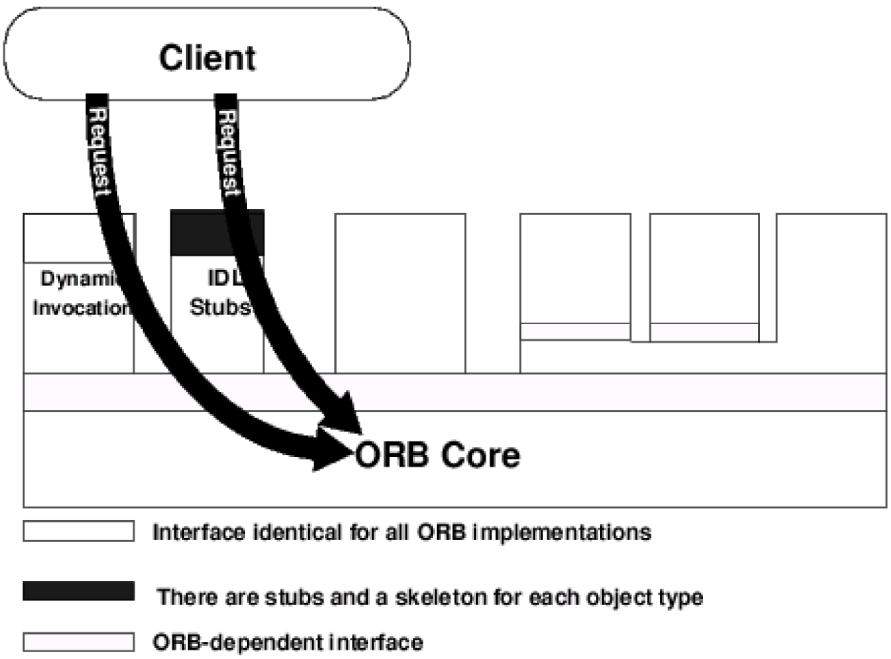


Figure 2-3 A Client Using the Stub or Dynamic Invocation Interface

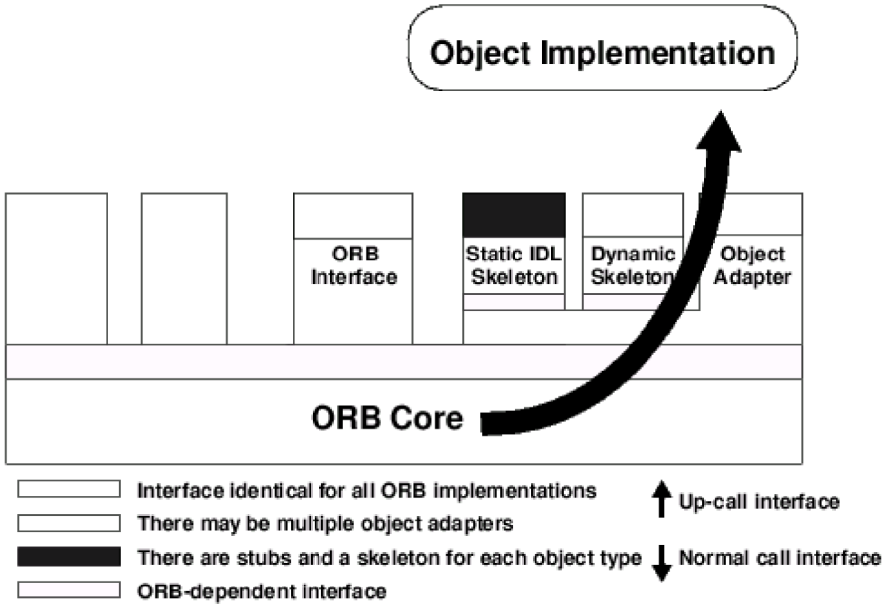


Figure 2-4 An Object Implementation Receiving a Request

Když se tedy dotazujeme například přes Trader službu na objekt s nějakými schopnostmi, často nemusíme znát jeho rozhraní. Při volání skrze stub musíme znát rozhraní objektu již v době kompilace aplikace. Nicméně při použití Dynamic Invocation Interface (DII) můžeme metody objektů volat pouze podle jmen, přičemž musíme pouze dodat správné parametry. Podobný trik můžeme používat i na straně serverového objektu, kdy nemusíme používat skeleton, ale můžeme skrze Dynamic Skeleton Interface (DSI) dekódovat jména volaných funkcí a jejich parametrů a obsluhovat je speciálním způsobem. Pro podporu těchto dynamických rozhraní je zde Interface Repository (IR), které se může aplikace dotazovat, pro získání popisu rozhraní jednotlivých CORBA objektů. Na obrázku 2-3 a 2-4 z [4] vidíme celkový pohled na proces volání funkce objektu.

Na obrázku 2-4 můžeme také vidět prvek, o kterém ještě nebyla zmínka – Object Adapter. Object Adapter je objekt zapouzdřující základní funkce správy objektu. Například poskytuje funkce pro uvolnění objektu z paměti, přičemž jeho stav je uložen na disk. Po té, až je objekt zase potřebný, pomáhá zase jeho stav obnovit. Všem procesům tohoto typu se říká transaparentní aktivace a deaktivace. Umožňuje například vytvářet objekty s délkou života mnohonásobně přesahující dobu běhu serveru mezi jeho restarty. Stručně řečeno Object Adapter se stará o objekt(y) jemu svěřené. Nejdůležitější přitom je, že v sobě skrývá systémově specifický kód a umožňuje tím snadnou přenositelnost objektů mezi jednotlivými implementacemi CORBy.

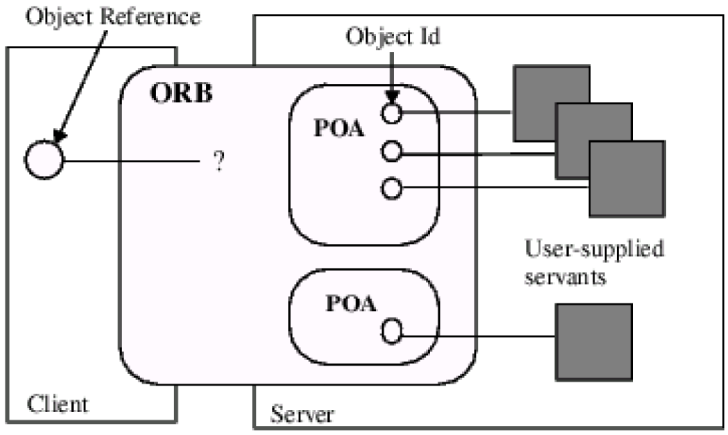
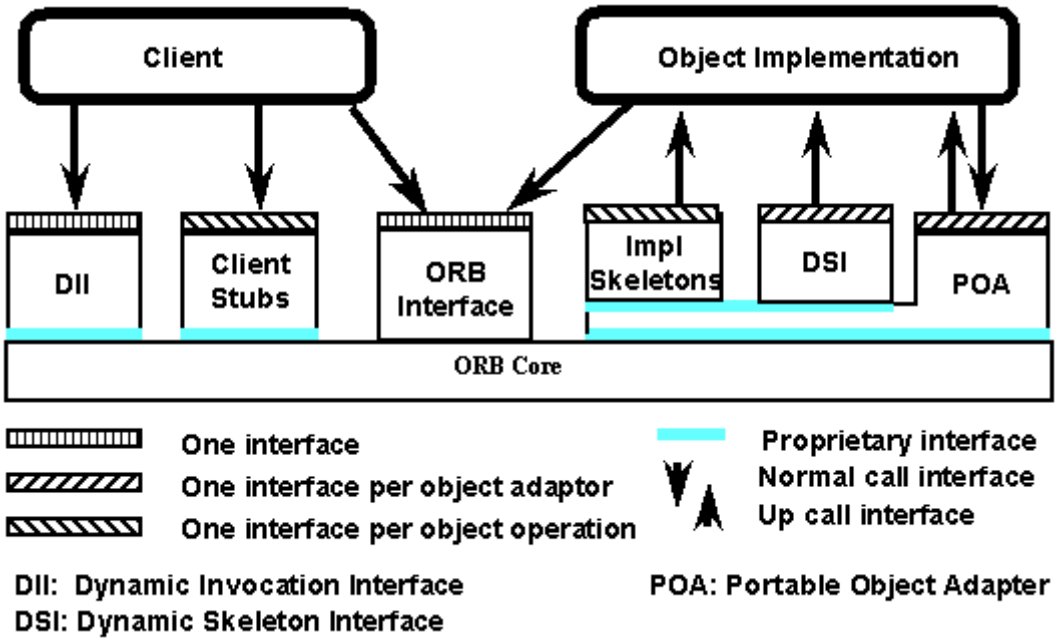


Figure 11-1 Abstract POA Model



Na tomto obrázku vidíme kompletní architekturu CORBy z pohledu jejího uživatele. Především jsou zde znázorněny, která rozhraní jsou standardizována a která ne. Jsou k dispozici snad i detailnější obrázky, ale ty jsou určeny programátorům samotných CORBA implementací a pro obyčejné lidi nemají přímý smysl.

Na závěr, jednoduchý příklad programu v CORBě. Jedná se o velmi jednoduchou aplikaci, která má člověka pouze uvést do problematiky. Příklad originálně pochází ze stránek knihovny ORBit, ze kterých jsem čerpal mnoho praktických myšlenek.

Program má dvě části – klientskou a serverovou. Klient přečte text ze stdin a server jej vypíše na svůj stdout.

Prvním úkolem pro programátora je specifikovat IDL rozhraní, které bude klient i server používat. V našem případě se jedná o jediný objekt s názvem Echo a jeho metodou echoString.

soubor echo.idl:

```
interface Echo {
	void echoString(in string input);
};
```

Zběhlého programátora může udivit deklarace in, která říká, že daný argument se předává pouze směrem k serveru.

Pro zkompilování IDL rozhraní použijeme příkaz: $ orbit-idl-2 --skeleton-impl echo.idl, který nám vygeneruje množství souborů, které by jsme mohli nazvat "framework".

File	Usage for Client	Usage for Server
echo.h	readonly	readonly
echo-common.c	readonly	readonly
echo-stubs.c	readonly	-
echo-skels.c	-	readonly
echo-skelimpl.c	-	template for user code
echo-client.c	client code	-
echo-server.c	-	generic code for servant creation

Nejdůležitější jsou pro programátora poslední dva soubory, které budou obyčejně těmi jedinými, které bude modifikovat.

echo-client.c source code:

```
/*
 * Echo client program.. Hacked by Ewan Birney <birney@sanger.ac.uk>
 * from echo test suite, update for ORBit2 by Frank Rehberger
 * <F.Rehberger@xtradyne.de>
 *
 * Client reads object reference (IOR) from local file 'echo.ior' and
 * forwards console input to echo-server. A dot . as single character
 * in input terminates the client.
 */
```

```
#include <stdio.h>
#include <signal.h>
#include <orbit/orbit.h>

/*
 * This header file was generated from the idl
 */

#include "echo.h"

/**
 * test for exception
 */
static gboolean raised_exception(CORBA_Environment *ev)
{
	return ((ev)->_major != CORBA_NO_EXCEPTION);
}

/**
 * in case of any exception this macro will abort the process
 */
static void abort_if_exception(CORBA_Environment *ev, const char* mesg)
{
	if (raised_exception (ev)) {
		g_error ("%s %s", mesg, CORBA_exception_id (ev));
		CORBA_exception_free (ev);
		abort();
	}
}

static CORBA_ORB  global_orb = CORBA_OBJECT_NIL;  /* global orb */

/* Is called in case of process signals. it invokes CORBA_ORB_shutdown()
 * function, which will terminate the processes main loop.
 */
static void client_shutdown (int sig)
{
	CORBA_Environment  local_ev[1];
	CORBA_exception_init(local_ev);

	if (global_orb != CORBA_OBJECT_NIL)
	{
		CORBA_ORB_shutdown (global_orb, FALSE, local_ev);
		abort_if_exception (local_ev, "caught exception");
	}
}


/* Inits ORB @orb using @argv arguments for configuration. For each
```

```
 * ORBit options consumed from vector @argv the counter of @argc_ptr
 * will be decremented. Signal handler is set to call
 * echo_client_shutdown function in case of SIGINT and SIGTERM
 * signals.  If error occures @ev points to exception object on
 * return.
 */
static void client_init (int *argc_ptr,
                         char *argv[],
                         CORBA_ORB *orb,
                         CORBA_Environment *ev)
{
        /* init signal handling */

        signal(SIGINT,  client_shutdown);
        signal(SIGTERM, client_shutdown);

        /* create Object Request Broker (ORB) */

        (*orb) = CORBA_ORB_init(argc_ptr, argv, "orbit-local-orb", ev);
        if (raised_exception(ev)) return;
}

/* Releases @servant object and finally destroys @orb. If error
 * occures @ev points to exception object on return.
 */
static void client_cleanup (CORBA_ORB           orb,
                            CORBA_Object        service,
                            CORBA_Environment *ev)
{
        /* releasing managed object */
        CORBA_Object_release(service, ev);
        if (raised_exception(ev)) return;

        /* tear down the ORB */
        if (orb != CORBA_OBJECT_NIL)
        {
                /* going to destroy orb.. */
                CORBA_ORB_destroy(orb, ev);
                if (raised_exception(ev)) return;
        }
}

/**
 *
 */
static CORBA_Object client_import_service_from_stream (CORBA_ORB           orb,
                                                       FILE                *stream,
                                                       CORBA_Environment *ev)
{
        CORBA_Object obj = CORBA_OBJECT_NIL;
```

```
	gchar *objref=NULL;

	fscanf (stream, "%as", &objref);  /* FIXME, handle input error */

	obj = (CORBA_Object) CORBA_ORB_string_to_object (global_orb,
	                                         objref,
	                                         ev);
	free (objref);

	return obj;
}

/**
 *
 */
static CORBA_Object client_import_service_from_file (CORBA_ORB             orb,
                                                     char                  *filename,
                                                     CORBA_Environment *ev)
{
	CORBA_Object  obj    = NULL;
	FILE         *file   = NULL;

	/* write objref to file */

	if ((file=fopen(filename, "r"))==NULL)
		g_error ("could not open %s\n", filename);

	obj=client_import_service_from_stream (orb, file, ev);

	fclose (file);

	return obj;
}


/**
 *
 */
static void client_run (Echo  echo_service,
                        CORBA_Environment *ev)
{
	char filebuffer[1024+1];

	g_print("Type messages to the server\n"
		"a single dot in line will terminate input\n");

	while( fgets(filebuffer,1024,stdin) ) {
		if( filebuffer[0] == '.' && filebuffer[1] == '\n' )
			break;
```

```
            /* chop the newline off */
            filebuffer[strlen(filebuffer)-1] = '\0';

            /* using the echoString method in the Echo object
             * this is defined in the echo.h header, compiled from
             * echo.idl */

            Echo_echoString(echo_service,filebuffer,ev);
            if (raised_exception (ev)) return;
        }
}

/*
 * main
 */
int main(int argc, char* argv[])
{
        CORBA_char filename[] = "echo.ior";

        Echo echo_service = CORBA_OBJECT_NIL;

        CORBA_Environment ev[1];
        CORBA_exception_init(ev);

        client_init (&argc, argv, &global_orb, ev);
        abort_if_exception(ev, "init failed");

        g_print ("Reading service reference from file \"%s\"\n", filename);

        echo_service = (Echo) client_import_service_from_file (global_orb,
                                                "echo.ior",
                                                ev);
        abort_if_exception(ev, "import service failed");

        client_run (echo_service, ev);
        abort_if_exception(ev, "service not reachable");

        client_cleanup (global_orb, echo_service, ev);
        abort_if_exception(ev, "cleanup failed");

        exit (0);
}
```

Tento program můžeme rozdělit na tři části:

- inicializace ORB
- získání CORBA objektu
- používání objektu

Klíčové místo programu je příkaz volání metody echoString. Toto volání je skrze ORB přeneseno až do serverovské aplikace. Z IDL definice v echo.idl

```
    void echoString(in string input);
```

bylo při kompilaci vytvořena v echo.h následující funkce:

```
extern void Echo_echoString(Echo            obj,
                            CORBA_char      *astring,
                            CORBA_Environment *ev);
```

C-čko samo o sobě pravé objekty nepodporuje, proto se problém částečně obchází obyčejnýma funkcema s jistými konvencemi, která se za léta zažila v praxi, tj. jméno objektu je předřazeno před vlastní název metody, první parametr je vlastní objekt a poslední parametr je struktura, která je určena k obsluze vyjímek. Toto je také systém, jak je CORBA mapována do jazyka C. V případě jazyka C++ bude situace samozřejmě jednodušší.

Echo Server

Server je více složitý, má ale mnoho společného s klientem. Takže kód nebude kompletně nový. Nejvíce času stráví server v hlavní smyčce aplikace, kde čeká na klienty. Do hlavní smyčky se server dostává voláním funkce CORBA_ORB_run(), která se v našem příkladu nachází ve funkci server_run(). Nejdříve se ale vytváří ORB, a pak CORBA objekty, které budeme zpřístupňovat klientům. Celý proces je poněkud složitější u realných aplikací, ale pro jednoduchý příklad to stačí.

echo-server.c source code:

```
/*
 * echo-server program. Hacked from Echo test suite by
 * %lt;birney@sanger.ac.uk>, ORBit2 udpate by Frank Rehberger
 * <F.Rehberger@xtradyne.de>
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <orbit/orbit.h>

#include "echo.h"
#include "echo-skelimpl.c"

/**
 * test for exception
 */
static gboolean raised_exception(CORBA_Environment *ev)
{
```

```
	return ((ev)->_major != CORBA_NO_EXCEPTION);
}

/**
 * in case of any exception this macro will abort the process
 */
static void abort_if_exception(CORBA_Environment *ev, const char* mesg)
{
	if (raised_exception (ev)) {
		g_error ("%s %s", mesg, CORBA_exception_id (ev));
		CORBA_exception_free (ev);
		abort();
	}
}

static CORBA_ORB  global_orb = CORBA_OBJECT_NIL;  /* global orb */

/* Is called in case of process signals. it invokes CORBA_ORB_shutdown()
 * function, which will terminate the processes main loop.
 */
static void server_shutdown (int sig)
{
	CORBA_Environment  local_ev[1];
	CORBA_exception_init(local_ev);

	if (global_orb != CORBA_OBJECT_NIL)
	{
		CORBA_ORB_shutdown (global_orb, FALSE, local_ev);
		abort_if_exception (local_ev, "caught exception");
	}
}

/* Inits ORB @orb using @argv arguments for configuration. For each
 * ORBit options consumed from vector @argv the counter of @argc_ptr
 * will be decremented. Signal handler is set to call
 * echo_server_shutdown function in case of SIGINT and SIGTERM
 * signals.  If error occures @ev points to exception object on
 * return.
 */
static void server_init (int  *argc_ptr,
			 char  *argv[],
			 CORBA_ORB  *orb,
			 CORBA_Environment  *ev)
{
	/* init signal handling */

	signal(SIGINT,  server_shutdown);
	signal(SIGTERM, server_shutdown);

	/* create Object Request Broker (ORB) */
```

```
        (*orb) = CORBA_ORB_init(argc_ptr, argv, "orbit-local-orb", ev);
        if (raised_exception(ev)) return;
}

/* Creates servant and registers in context of ORB @orb. The ORB will
 * delegate incoming requests to specific servant object.  @return
 * object reference. If error occures @ev points to exception object
 * on return.
 */
static Echo server_activate_service (CORBA_ORB              orb,
                                     CORBA_Environment *ev)
{
        Echo                    servant     = CORBA_OBJECT_NIL;
        PortableServer_POA      poa         = CORBA_OBJECT_NIL;
        PortableServer_POAManager  poa_manager = CORBA_OBJECT_NIL;

        /* get Portable Object Adaptor (POA) */

        poa =
         (PortableServer_POA) CORBA_ORB_resolve_initial_references(orb,
                                              "RootPOA",
                                              ev);
        if (raised_exception(ev)) return CORBA_OBJECT_NIL;

       /* create servant in context of poa container */

        servant = impl_Echo__create (poa, ev);
        if (raised_exception(ev)) return CORBA_OBJECT_NIL;

        /* activate POA Manager */

        poa_manager = PortableServer_POA__get_the_POAManager(poa, ev);
        if (raised_exception(ev)) return CORBA_OBJECT_NIL;

        PortableServer_POAManager_activate(poa_manager, ev);
        if (raised_exception(ev)) return CORBA_OBJECT_NIL;

        return servant;
}

/* Writes stringified object reference of @servant to file
 * @filename. If error occures @ev points to exception object on
 * return.
 */
static void server_export_service_to_file (CORBA_ORB            orb,
                                           CORBA_Object         servant,
                                           char                 *filename,
                                           CORBA_Environment *ev)
{
```

```
        CORBA_char *objref = NULL;
        FILE       *file   = NULL;

        /* write objref to file */

        objref = CORBA_ORB_object_to_string (orb, servant, ev);
        if (raised_exception(ev)) return;

        if ((file=fopen(filename, "w"))==NULL)
                g_error ("could not open %s\n", filename);

        /* print ior to terminal */
        fprintf (file, "%s\n", objref);
        fflush (file);
        fclose (file);

        CORBA_free (objref);
}

/* Entering main loop @orb handles incoming request and delegates to
 * servants. If error occures @ev points to exception object on
 * return.
 */
static void server_run (CORBA_ORB             orb,
                        CORBA_Environment *ev)
{
        /* enter main loop until SIGINT or SIGTERM */

        CORBA_ORB_run(orb, ev);
        if (raised_exception(ev)) return;

        /* user pressed SIGINT or SIGTERM and in signal handler
         * CORBA_ORB_shutdown(.) has been called */
}

/* Releases @servant object and finally destroys @orb. If error
 * occures @ev points to exception object on return.
 */
static void server_cleanup (CORBA_ORB           orb,
                            Echo                servant,
                            CORBA_Environment *ev)
{
        /* releasing managed object */
        CORBA_Object_release(servant, ev);
        if (raised_exception(ev)) return;

        /* tear down the ORB */
        if (orb != CORBA_OBJECT_NIL)
        {
                /* going to destroy orb.. */
```

```
            CORBA_ORB_destroy(orb, ev);
            if (raised_exception(ev)) return;
        }
}

/*
 * main
 */

int main (int argc, char *argv[])
{
        Echo servant = CORBA_OBJECT_NIL;

        CORBA_Environment  ev[1];
        CORBA_exception_init(ev);

        server_init (&argc, argv, &global_orb, ev);
        abort_if_exception(ev, "init failed");

        servant = server_activate_service (global_orb, ev);
        abort_if_exception(ev, "activating service failed");

        server_export_service_to_file (global_orb,
                                       servant,
                                       "echo.ior",
                                       ev);
        abort_if_exception(ev, "exporting IOR failed");

        server_run (global_orb, ev);
        abort_if_exception(ev, "entering main loop failed");

        server_cleanup (global_orb, servant, ev);
        abort_if_exception(ev, "cleanup failed");

        exit (0);
}
```

Klíčová část této server aplikace je volání "servant = impl_Echo__create (poa, ev);". Tato funkce je definována v echo-skelimpl.c a je includována na začátku našeho echo-server.c. Pro každou metodu echo objektu je zde předdefinován kód, který musí uživatel doplnit. Podle přicházejících požadavků object manager volá metody objektu – v našem příkladu je přítomná pouze jediná metoda echoString(). Když se podíváme do souboru echo-skelimpl.c, kterou pro nás vygeneroval orbit-idl-2 nebo nastroj jiné implementace CORBy. Jediný řádek je potřeba uživatelem doplnit – samotný kód obsluhující metodu echoString. Do této funkce, která je až na samém konci souboru, je vložen kód g_print ("%s\n", input);.

Konstruktor (create) a destruktor (destroy) jsou zde definovány také. Pokud nemáme speciální požadavky na objekt, mohou zůstat nezměněny.

echo-skelimpl.c source code:

```
#include "echo.h"

/*** App-specific servant structures ***/

typedef struct
{
  POA_Echo servant;
  PortableServer_POA poa;

  /* ------ add private attributes here ------ */
  /* ------ ---------- end ------------ ------ */
}
impl_POA_Echo;

/*** Implementation stub prototypes ***/

static void impl_Echo__destroy(impl_POA_Echo * servant,
                               CORBA_Environment * ev);
static void
impl_Echo_echoString(impl_POA_Echo * servant,
                     const CORBA_char * input, CORBA_Environment * ev);

/*** epv structures ***/

static PortableServer_ServantBase__epv impl_Echo_base_epv = {
  NULL,                       /* _private data */
  (gpointer) & impl_Echo__destroy,     /* finalize routine */
  NULL,                       /* default_POA routine */
};
static POA_Echo__epv impl_Echo_epv = {
  NULL,                       /* _private */
  (gpointer) & impl_Echo_echoString,

};

/*** vepv structures ***/

static POA_Echo__vepv impl_Echo_vepv = {
  &impl_Echo_base_epv,
  &impl_Echo_epv,
};

/*** Stub implementations ***/

static Echo
```

```
impl_Echo__create(PortableServer_POA poa, CORBA_Environment * ev)
{
  Echo retval;
  impl_POA_Echo *newservant;
  PortableServer_ObjectId *objid;

  newservant = g_new0(impl_POA_Echo, 1);
  newservant->servant.vepv = &impl_Echo_vepv;
  newservant->poa =
    (PortableServer_POA) CORBA_Object_duplicate((CORBA_Object) poa, ev);
  POA_Echo__init((PortableServer_Servant) newservant, ev);
  /* Before servant is going to be activated all
   * private attributes must be initialized.  */

  /* ------ init private attributes here ------ */
  /* ------ ---------- end ------------- ------ */

  objid = PortableServer_POA_activate_object(poa, newservant, ev);
  CORBA_free(objid);
  retval = PortableServer_POA_servant_to_reference(poa, newservant, ev);

  return retval;
}

static void
impl_Echo__destroy(impl_POA_Echo * servant, CORBA_Environment * ev)
{
  CORBA_Object_release((CORBA_Object) servant->poa, ev);

  /* No further remote method calls are delegated to
   * servant and you may free your private attributes. */
  /* ------ free private attributes here ------ */
  /* ------ ---------- end ------------- ------ */

  POA_Echo__fini((PortableServer_Servant) servant, ev);
}

static void
impl_Echo_echoString(impl_POA_Echo * servant,
                     const CORBA_char * input, CORBA_Environment * ev)
{
  /* ------   insert method code here   ------ */
  g_print ("%s\n", input);
  /* ------ ---------- end ------------ ------ */
}
```

Další informace ohledně tohoto příkladu jsou v [6].

Literatura:

[1] OMG stránky, http://www.omg.org/

[2] OMG, *CORBA Basics,*
http://www.omg.org/gettingstarted/corbafaq.htm

[3] OMG, *ORB Basics,*
http://www.omg.org/gettingstarted/orb_basics.htm

[4] OMG, *CORBA version 3.0*, July 2002,
přístupno skrz vyhledávač na http://www.omg.org/cgi-bin/apps/doclist.pl

[5] ORBit team, *ORBit Beginners Documentation V1.2*,
http://www.gnome.org/projects/ORBit2/orbit-docs/orbit/book1.html

[6] ORBit team, *Echo client & server*,
http://www.gnome.org/projects/ORBit2/orbit-docs/orbit/x278.html

[7] *CORBA in five minutes*,
http://ww.telent.net/corba/

[8] GNOME team, *CORBA, ORBit and Bonobo*,
http://canvas.gnome.org:65348/gnomefaq/html/x703.html

[9] Michael Meeks, *Introduction to Bonobo: Bonobo & ORBit*, 2001,
http://www-106.ibm.com/developerworks/webservices/library/co-bnbo1.html

[10] ORBit team, *ORBit FAQ*,
http://orbit-resource.sourceforge.net/faq.html

[11] Kate Keahey, *A Brief Tutorial on CORBA*,
http://www.cs.indiana.edu/~kksiazek/tuto.html

[12] OMG, *Interoperable Naming Service Specification*, 2000,
přístupno skrz vyhledávač na http://www.omg.org/cgi-bin/apps/doclist.pl

[13] OMG, *Trading Object Service Specification*, 1997
přístupno skrz vyhledávač na http://www.omg.org/cgi-bin/apps/doclist.pl

[13] Linas Vepstas, *Linux DCE, CORBA and DCOM Guide* (sbírka všech možných implementací CORBy a podobných systémů), http://linas.org/linux/corba.html