UNIVERZITA J. E. PURKYNĚ V ÚSTÍ NAD LABEM
PEDAGOGICKÁ FAKULTA

DISTRIBUOVANÉ OBJEKTOVÉ SYSTÉMY
—
CORBA

Katedra informatiky

Vedoucí bakalářské práce : Mgr. Jiří Fišer Ph.D.
Autor bakalářské práce : Michal Duda
Studijní obor : Informační systémy

Datum dokončení bakalářské práce: duben 2003

Rád bych poděkoval panu Mgr. Jiřímu Fišerovi Ph.D. a všem svým pedagogům za obětavou pomoc, cenné rady a vedení mé bakalářské práce.

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně, použil pouze literatury uvedené v seznamu na konci této bakalářské práce a veškeré použité citace jsem řádně označil.

V Roudnici nad Labem dne ..... podpis .....

Shrnutí

Ve své bakalářské práci se zabývám technologií CORBA a jejími alternativami. Cílem práce je ukázat jednoduché použítí této technologie v operačním systému Linux za pomoci programovacího jazyka C++. Bude zde popsána funkčnost a základní programové rozhraní modelu CORBA. V práci nebude brán zřetel na specifikace skupiny OMG nižší než verze 2.3. Základním stavebním kamenem by se zde měla stát norma verze 2.3, přičemž se budu snažit obohatit tuto práci o možnosti nabízené novějšími specifikacemi.

Terminologie

Vzhledem k tomu, že distribuované objektové systémy nemají prozatím zavedenou terminologii, budu zde používat terminologii převzatou z knihy „COM+, CORBA, EJB“[DOB-01].

Zkratky

BOA — Basic Object Adapter
CORBA — Common Object Request Broker Architecture
COSS — Common Object Servicies Specification
DCOM — Distributed Common Object Model
DCOP — Desktop COmunication Protocol
DII — Dynamic Invocation Interface
DSI — Dynamic Skeleton Interface
EJB — Enterprise JavaBeans
GIOP — General Inter–ORB Protocol
IDL — Interface Description Language
IIOP — Internet Inter–ORB Protocol
IOR — Interoperable Object Reference
OMG — Object Management Group
ORB — Object Request Broker
POA — Portable Object Adapter
RMI — Remote Method Invocation
RPC — Remote Procedure Call
SII — Static Invocation Interface
SOAP — Simple Object Access Protocol
SSI — Static Skeleton Interface

Značení

písmo	význam
`text`	programové konstrukce, termíny, názvy
text	název v původním jazyce

Zvýraznění textu

označení	význam
text	klíčová slova
`text`	parametr za nějž dosadíme konkrétní hodnotu
`[ text ]`	nepoviný údaj
`...`	libovolný počet výskytů
`v1 | v2`	v1 nebo v2

Zjednodušená Backus–Naurova forma zápisu

Obsah

Seznam výpisů

Předmluva

V současné době se začíná stále více využívat objektově orientovaného přístupu při vývoji aplikací. První vývojový nástroj založený na objektovém přístupu se zřejmě poprvé objevil v roce 1966, jednalo se o jazyk Simula. Sice ještě nešlo o jazyk splňující všechny předpoklady dnešních objektových jazyků, ale objevil se zde poprvé koncept třídy. V roce 1970 vznikl zatím asi „nejčistší“ objektový nástroj Smalltalk. Zde se již objevil koncept dědičnosti. O deset let později se začíná používat, asi dnes jeden z nejznámějších jazyků, C++. Tak by se dal stručně charakterizovat vývoj objektových nástrojů. Stále zůstává nezodpovězena otázka, proč právě používat objektově orientovaný přístup při vývoji. Na tuto otázku lze nalézt dvě nejzákladnější odpovědi. První z nich je ta, že některé problémy se dají řešit lépe pomocí objektového modelu. Zpřehlední se tak zdrojový kód a zároveň se zjednoduší i celkový pohled na funkčnost aplikace. Je to dáno díky tomu, že problém můžeme rozdělit na dílčí části, které můžeme reprezentovat jako objekty. Pro reprezentaci objektů a jejich vzájemných závislostí slouží skupina popisných jazyků, mezi nejznámější patří například jazyk UML. Zde musím podotknout, že výraz „celkový pohled na funkčnost aplikace“ nelze automaticky chápat z pohledu stavů objektů, ale pouze z pohledu vzájemného vztahu objektů obsažených v aplikaci. Druhá odpověď by se dala nazvat jedním slovem znovupoužitelnost. Vytváření aplikací vyžaduje značné programátorské úsilí a velkou časovou zátěž. Obojí s rostoucím množstvím zvyšuje finanční náklady, a tak se nabízí možnost vytvářet takové objekty, aby byly znovupoužitelné. Tím se eliminuje čas i lidské zdroje na vytváření něčeho, co již bylo někdy dříve vytvořeno. Znovupoužitelnost se stává poslední dobou i předmětem obchodu.

Zhruba v 97 roce 20. století začíná období[1], kdy se objevují technologie umožňující používat objektový přístup distribuovaně. Mezi takovou technologii lze zařadit technologii CORBA.

[1] toto období se někdy označuje jako *intergalactic era*

Kapitola 1

Úvod

Technologii CORBA lze zařadit do oblasti middleware. Je spravovaná organizací OMG založenou velkými společnostmi působících v oblasti informatiky. Použití této technologie není moc jednoduchá záležitost a využívá se především v rozsáhlých projektech jako jsou: aplikace pro vědeckotechnické výpočty, bankovní aplikace, aplikace užité ve vojenské sféře, apod.. V současné době existuje velké množství implementací různých dodavatelů, lišících se od podporovaných operačních systémů a hardwarových platforem až po rozličné programovací jazyky. Většina podporovaných jazyků je objektová, jelikož se jedná o technologii, která je postavena na využívání distribuovaných objektů. Existuje ale i podpora neobjektových jazyků, jako je například ANSI C. Zde se ale objektový přístup pouze simuluje a musí se uchovávat stav použitých pseudoobjektů ve zvláštních proměnných. Právě možnost využití různých softwarových či hardwarových platforem činí z CORBy velmi silný nástroj. Avšak příměrem k tomu je to někdy spíše na škodu, než k užitku. Důležitým a nesnadným rozhodnutím je určení, zda je tato technologie nejvhodnější. Obecně by při výběru měl být brán v potaz, zda použitá technologie by měla být velmi flexibilní a univerzální nebo jestli by spíše měla být více specializovaná. Tato otázka by měla být zodpovězena před začátkem návrhu projektu. V souvislosti s tímto problémem se budu snažit srovnat různé distribuované architektury.

V druhé kapitole této práce se zabývám vývojem distribuovaných systémů a pohledem na ně od počátků až k současnosti. Třetí část je věnována základnímu popisu vlastností technologie CORBA a jejím alternativám. Tyto technologie byly zvoleny pro svoji velkou rozšířenost. Další část zmiňuje základní stavbu architektury CORBA. Jsou zde uveřejněny informace o základním stavebním kamenu ORB, o popisném jazyku OMG IDL, komunikačním a objektovém modelu a detailněji o standardizovaných objektových službách. Pátá kapitola uvádí podrobnější popis pro jazyk OMG IDL a základní programově aplikační rozhraní CORBA. V předposlední kapitole se zabývám implementací testovací aplikace, její funkčností a jejím návrhem. Na ní bude vysvětleno rozhraní, které nebylo popsáno v předchozí kapitole o rozhraní.

Kapitola 2

Distribuované objektové systémy

Již od samých začátků použití počítačových sítí se objevuje snaha centralizovat služby v uceleném stavu, aby bylo možno sdílet data a výpočetní zdroje. Tato kapitola se snaží poskytnout náhled na historický vývoj programátorských stylů.

2.1 monolitická architektura

První architektura použitá pro centralizaci se označuje pojmem monolitická. Lze ji považovat za historicky nejstarší. Byla využívána v době, kdy byly k disposici relativně drahé počítačové komponenty. V takových počítačových sítích byl umístěn jeden silný výpočetní prvek označovaný jako *mainframe*. K němu byly připojeny terminály, které byly výrazně levnější. Terminály umožňovaly uživateli interakci se softwarem umístěným na mainframu. Jelikož terminál neměl žádné výpočetní zdroje, přenášel všechny vstupní operace do mainframu, kde byly následně zpracovány. Software na mainframu byl monolitického charakteru. Všechny komponenty aplikace, konkrétně uživatelské prostředí, aplikační logika a databázové úložiště, tvořily jeden celek. Pokud byl software dobře navržen, byl schopen obsloužit velké množství uživatelů. Jeho nevýhodou však byla prakticky nemožnost postupného rozšiřování.

2.2 klient/server architektura

S poklesem cen výpočetních zdrojů na únosnou míru, se začala objevovat nová architektura nazvaná klient/server. Systém založený na této architektuře již neobsahuje terminály, ale je složen z několika plnohodnotných počítačů označovaných jako klienti a jedné výpočetně výkonné jednotky nazývané termínem server. Z původně monolitického

systému se oddělilo uživatelské rozhraní a přesunulo se ve formě aplikace na klienty. Tím byla snížena náročnost na zpracování vstupu na centrálním softwaru, zde se již všechny vstupní operace vyhodnotily a na server byly poslány jenom žádosti o poskytnutí konkrétních služeb.

Postupem času byla navrhnuta vícevrstvá architektura typu klient/server. Nejznámnější z ní je dvouvrstvá a třívrstvá verze.

2.2.1 dvouvrstvá architektura

Dvouvrstvá architektura se především využívá v menších projektech. Mezi její hlavní omezení patří:

1. Počet možných obsloužených uživatelů, jedná se řádově o stovky.
2. Nemožnost dosáhnutí nějaké výkonné zabezpečovací politiky.
3. Špatná rozšiřitelnost.

Naproti tomu výhoda je patrná pouze jedna. Pokud bude aplikační logika umístěna v serverové části, bude dobře využita jako znovupoužitelný kód a zároveň bude snadnější správa celého systému.

2.2.2 třívrstvá architektura

Dalším krokem bylo přidání třetí vrstvy. Nyní již první vrstvu tvoří grafické rozhraní umístěné na klientu, další vrstva obsahuje aplikační logiku a neměnná data. Poslední vrstvu tvoří obecné služby jako jsou databázová úložiště, transakční monitory a jiné.

Tyto tři vrstvy mají své názvy:

1. vrstva prezentační
2. vrstva aplikační
3. vrstva datová

Vrstva prezentační komunikuje pouze s vrstvou aplikační, ale nikdy nemůže přistupovat přímo k vrstvě datové. Aplikační vrstva může komunikovat jak s uživatelským rozhraním v prezentační vrstvě, tak i s datovou vrstvou. Toto úzké hrdlo, tvořené aplikační logikou, umožňuje snadnou údržbu celého systému. Mezi další výhody náleží:

1. Možnost výměny nějaké části z vrstvy datové za jinou, aniž by byla zasažena funkčnost klienta.
2. Aplikační logika může být rozdělena ve svých kopiích do více počítačů pro vyvážení zátěže, což u dvouvrstvé architektury nebylo možné.
3. Lze dosáhnout vysokého stupně zabezpečení z důvodu oddělení datové části od aplikační.

2.3 distribuované objekty

Logickým vyústěním architektury klient/server jsou distribuované objektové systémy. Ty se začaly objevovat se stále se zvětšujícím zájmem o objektový způsob vývoje programů. Na rozdíl od architektury klient/server zde není brán v potaz funkční pohled na části aplikace, ale používají se jiné logické celky, tzv. objekty. U těchto systémů jsou využity všechny základní vlastnosti objektově orientovaného způsobu programování.

1. Zapouzdřenost — umožňuje nezávislý vývoj jednotlivých objektů na jiných objektech, aniž by muselo být známo, jakým způsobem pracují. Vývoj se provádí na základě předem stanoveného rozhraní jednotlivých objektů.
2. Polymorfizmus a dědičnost — díky pozdní vazbě lze využívat služeb specializovaných objektů neznámých v době překladu.

Tato architektura s sebou přinesla velkou flexibilitu celého systému, jak již při vývoji, tak i při správě.

Kapitola 3

Distribuované objekty — alternativy

3.1 RPC

Technologie RPC *(Remote Procedure Call)* nepoužívá objektový přístup. Jak již název napovídá tato metoda je založena na distribuování procedur. Původně byla navržena společností Sun Microsystems při vývoji protokolu NFS. Později byla přijata pod názvem ONC RPC *(Open Network Computing RPC)*. Tato architektura byla prvním úspěšným pokusem o přiblížení dosud používaného stylu síťového programování co možná k nejvíce běžnému způsobu vývoje nesíťových aplikací. Celá architektura je postavena na transportních protokolech TCP a UDP. Programátor se již nemusí zajímat o hardwarovou architekturu, jelikož se o přenos dat starají rutiny standardu XDR *(eXtended Data Representation)*. Tyto rutiny jsou generovány z popisu rozhraní aplikace, které bude sdíleno. Dále je vygenerována kostra programu, jež zajistí správné volaní procedur a použití kódu XDR. Značnou nevýhodou RPC je možnost předávání pouze jednoho parametru u každé procedury. Případné větší množství parametrů se musí uspořádat do struktury a tu následně předat jako parametr. Každý RPC server má přidělené jednoznačné číslo, které je dáno určitými algoritmy. To je pak registrováno při spuštění serveru u služby `portmap`. Jelikož tato služba má vyhrazený port 111, klientská aplikace je schopna vždy navázat se serverem spojení pomocí služby `portmap`. Ta převede jednoznačné číslo na soket, kde naslouchá požadovaný server.

Model RPC nepředstavuje bezpečnou službu, protože všechny informace o něm posílané po síti se přenášejí v textové podobě. Nicméně alespoň podporuje několik ověřovacích mechanizmů.

1. implicitně je použita volba neověřování identity
2. ověření je provedeno pomocí prověřovacích metod operačního systému Unix (UID a GID)

3. ověření pomocí algoritmu DES
4. ověření na základě klíčů modulu Kerbeos

Ať již je RPC bezpečná, či nikoli, je použita jako základní stavební prvek pro další distribuované systémy. Existují zde ovšem jisté nekompatibility, které jsou způsobeny různorodou implementací. Příkladem může být technologie DCE RPC společnosti Microsoft. Na ní je založena dále popisovaná technologie COM.

3.2 COM, DCOM, COM+

COM *(Component Object Model)* je proprietární technologie firmy Microsoft, která vznikla návrhem nové verze mechanizmu sdílení dokumentů OLE 2.0 *(Object Linking and Embedding)*. OLE 2.0 byl navržen pro větší univerzálnost a robustnost, než měl jeho předchůdce. Původní realizace OLE nebyla vystavěna na objektově orientovaném modelu. Tato technika sdílení dokumentů byla příliš složitá a nebyla dobře přijata programátorskou komunitou. Pro zjednodušení použití a zvýšení efektivnosti při vývoji byl mechanizmus OLE kompletně předělán tak, aby vyhovoval objektově orientovanému přístupu. Při návrhu byla oddělena knihovna funkcí pro vkládání dokumentů od mechanizmu vzájemné komunikace programů. Takto osamostatněný model komunikace se nazývá COM.

Technologie COM vznikla ještě za dob 16bitových Windows 3.11. Tato nadstavba DOSu spravovala procesy v jednotném adresovém prostoru. Po přechodu na 32bitová Windows byl využit model správy paměti doposud známý z operačního systému UNIX. Každý spuštěný proces má vlastní adresový prostor. Návrh COMu s přechodem na takový přístup správy prostředků počítal a je vybaven konstrukcemi, kterými lze dosáhnout vzájemné komunikace mezi procesy v oddělených adresových prostorách. Přístup k těmto dvěma technikám správy se liší. Jednodušší přístup k objektům COM je pro procesy běžící ve společném adresovém prostoru alespoň co se týče C++ nebo jiných relativně nízkoúrovňových jazyků.

Specifikace COM definuje tři důležité termíny:

1. rozhraní — popis nabízených služeb
2. třída — zapouzdřená entita, která implementuje rozhraní
3. objekt — instance třídy
4. komponenta — obsahuje binární kód implementace třídy nebo více tříd

Konečná binární podoba komponenty má dva tvary:

- forma dynamicky linkované knihovny
- forma spustitelného souboru

Nejjednodušší případ, který může nastat, je při používání komponenty procesem ve svém adresovém prostoru. K tomu slouží tvar dynamicky linkované knihovny. Tu proces načte do svého adresového prostoru. Komponenta nezabírá paměť déle než je nezbytně nutné. Po skončení činnosti je opět uvolněna z paměti.

Složitější situace vzniká při použití komponenty v odděleném adresovém prostoru. K tomu se využívá spustitelný soubor, ve kterém je komponenta umístěna. Zde již programátor musí počítat s použitím generované kostry programu. Kostra se skládá ze zástupného *(proxy)* objektu na straně klienta, a `stubu` na straně serveru.

Aby mohly procesy komunikovat s objekty, musí být stanoven přesný binární tvar objektu a jeho rozhraní. V jazyku C++ je toto dosaženo pro rozhraní pomocí čistě abstraktních virtuálních tříd. Samotná COM třída je vytvořena za využití vícenásobné dědičnosti.

Na stejném principu pracují i technologie DCOM a COM+. DCOM se poprvé objevila spolu s operačním systémem Windows NT. Pro tuto verzi COMu byla charakteristická možnost distribuovat komponenty již za hranice operačního systému. Zatím nejnovější verzí je COM+. Ta byla poprvé uveřejněna s Windows 2000.

3.3 RMI

Architektura RMI *(Remote Method Invocation)* je první technologií umožňující distribuovat objekty pomocí programovacího jazyka Java. Poprvé byla do něj zahrnuta od verze 1.1. Ve svých počátcích byla RMI schopna pracovat se vzdálenými objekty, založenými pouze na Javě. Od verze 1.3 se však změnil dosud používaný komunikační protokol a začal se využívat protokol IIOP definovaný organizací OMG. Tím byla umožněna spolupráce více distribuovaných objektových systémů najednou. Použitím protokolu IIOP se tak dosáhlo velké otevřenosti, protože existuje mnoho specifikací, které dokáží zaručit přemostění mezi tímto a dalšími protokoly.

Podobně jako u architektur COM nebo CORBA je i zde vytvářena kostra programu. Problém může nastat tehdy, chceme-li použít rozhraní definované pro jinou architekturu než je RMI. V takovém případě již nelze použít překlad z jazyka IDL, ale musí se celé přepsat ručně do Javy. Příčinou je odlišný způsob vytváření kostry programu.

Na rozdíl od technologie CORBA je RMI značně jednodušší. Jsou zde využity některé vlastnosti jazyka Java. Příkladem může být serializace objektů. Ta je standardně definovaná pro základní typy a pro ostatní typy může být lehce odvozena implementací příslušného rozhraní. Serializace je pak využita pro přenos dat mezi serverem a klientem.

3.4 EJB

Po uvedení RMI do praxe se v roce 1998 začíná objevovat nová technologie EJB *(Enterprise JavaBeans)*. Podobně jako u architektury CORBA je zde kladen důraz na předem definované služby, které usnadní vývoj distribuovaných aplikací. Následkem toho již mohou aplikace, respektive objekty, také využívat transakčního zpracování, zabezpečení a zprávy životního cyklu, podpory vícevláknového přístupu. Všechny výše jmenované komponenty jsou umístěny v kontejneru objektů EJB. Ten slouží jako prostředí zpracování. Jeho úloha je pevně stanovena, čímž je zaručena nezávislost implementace. Pro přístup k vlastněným komponentám jsou definována přesná rozhraní, přičemž je tak opět dosaženo nezávislosti implementace na dodavateli. Vnitřně komunikace se vzdálenými objekty probíhá na základě architektury RMI.

Specifikace EJB definuje dvě hlavní skupiny objektů:

1. relační objekty *(session beans)* — zastupují perzistentní data. Obvykle jeden objekt odpovídá jednomu záznamu v databázové tabulce. EJB zajistí mapování instancí jednoho objektu na celou tabulku.

2. entitní objekty *(entity beans)* — poskytují služby klientským aplikacím. Lze je rozdělit na stavové a bezstavové. Stavové objekty uchovávají během své životnosti svůj stav, na rozdíl od bezstavových objektů, které lze přirovnat ke statickým metodám v jazyce C++.

Výhodou, jak architektury EJB tak i RMI, může být i velikost výsledného kódu *(bajt-kód)*.

3.5 SOAP

SOAP *(Simple Object Acess Protocol)* je nízkoúrovňový protokol založený na bázi značkovacího jazyku XML *(eXtended Markup Language)*. Základní podnět k jeho vzniku nastal při snaze přiblížit distribuovaný sytém co nejvíce k rozšířenému použití služeb poskytovaných sítí Internet. Na rozdíl od všech zde zmiňovaných systémů je SOAP jediný, který se snaží zachovat efektivnost a jednoduchost při jeho použití.

SOAP přenáší zprávy nejčastěji pomocí aplikačního protokolu HTTP. Zprávy jsou ve své podstatě XML dokumenty obsahující kořenový element `Envelope`. Ten je rozdělen na hlavičky `Header` a tělo `Body`. Tělo obsahuje zprávu určenou pro koncového příjemce, zatímco hlavička nese informaci pro různé prostředníky komunikace. Hlavní nevýhodou může být asynchronnost předávání zpráv, což může u řady aplikací přivodit složitost návrhu. Synchronního přenosu se dá dosáhnout jazykem WSDL *(Web Services Description Language)*. Zajímavou vlastností SOAPu je možnost dosažení pouze částečné synchronnosti pomocí jazyka WSFL *(Web Services Flow Language)*. Pomocí SOAP se dá vytvořit například i přemostění mezi architekturami COM a CORBA. Určitou zvláštností, plynoucí z použití aplikačních protokolů rodiny TCP/IP, je vyřešení průchodu zprávy skrze většinu firewallů.

3.6 DCOP a Gnorba

DCOP *(Desktop COmunication Protocol)* a Gnorba jsou zástupci zvláštní skupiny technologií. Jedná se o architektury, které jsou zbaveny velké a pro jejich užití zbytečné komplexnosti technologie CORBA. DCOP a Gnorba jsou využity v grafických manažerech, používaných na operačním systému Linux.

Gnorba je použita v projektu GNOME. Pomocí ní je vytvořena vrstva, která je umístěná mezi implementací ORBit modelu CORBA a komponentovým systémem Bonobo. Architektura DCOP je používána v manažeru KDE. Původně byla pro KDE použita implementace MICO architektury CORBA. Po necelých dvou letech je vyvinuta technologie DCOP. Její návrh byl proveden na zkušenostech z dosavadního používání MICO implementace. Ve skutečnosti jde o přesně opačný druh vývoje než je u Gnorby. Ta se soustředila na vytvoření jednoduššího rozhraní, přes které je možno komunikovat s CORBou. U DCOP se převzaly pouze teoretické výhody CORBy a celá technologie byla znovu implementována s využitím komunikačního protokolu ICE.

3.7 CORBA

Jelikož CORBA je hlavní náplní této práce, tak zde uvedu pouze přehled hlavních vlastností rozčleněných na výhody a nevýhody. Podrobnější výklad bude učiněn v následujících kapitolách.

Výhody:

- nezávislost na operačním systému
- nezávislost na hardwaru

- nezávislost na jazyku
- robustnost
- rozhraní distribuované části je popsáno jazykem IDL, odpadá psání deklarací v konkrétním jazyce
- použití protokolu GIOP, jednotný nízkoúrovňový protokol, nad kterým může stát libovolný síťový protokol
- standardizace protokolu IIOP, který je umístěn mezi protokoly GIOP a TCP/IP
- existuje mnoho specializovaných verzí, například RealTime CORBA
- podpora dynamického zjištění dostupnosti objektů až za běhu aplikace
- existence mnoha implementací od různých výrobců s bezproblémovou komunikací
- neustálý vývoj

Nevýhody:

- vysoké ceny při vývoji, někdy způsobené kvůli licenčním podmínkám
- robustnost a komplexnost někdy na škodu

Kapitola 4

CORBA — základní struktura

4.1 základní pojmy

Předem než začnu s popisem jednotlivých částí technologie CORBA, uvedu zde základní termíny, s kterými budu dále pracovat.

Základním stavebním kamenem každého objektového systému je `objekt`. Můžeme ho definovat jako identifikovatelnou zapouzdřenou entitu, která poskytuje jednu či více služeb. Aby bylo možné k objektu přistupovat, je k němu zapotřebí přiřadit jednoznačný identifikátor tzv. `referenci na objekt` *(object reference)*. Objekty mohou být libovolně vytvářeny nebo destruovány. V případě konstrukce objektu je vytvořená reference na objekt, která se potom šíří distribuovaným prostředím jako zástupce vzdáleného či lokálního objektu. Objekt zpřístupňuje své služby svému okolí pomocí jím definovaného `rozhraní`. Rozhraní definuje jakým způsobem mohou být požadavky na služby volány. Dva objekty jsou pak totožné pokud implementují stejné rozhraní. Služby poskytované objektem se nazývají `operace`. Operace je identifikovatelná entita a je složena ze:

- seznamu parametrů
- návratové hodnoty
- seznamu výjimek vyvolaných při chybě
- kontextové informace
- způsobu volání

Celý objektový systém se dělí na dvě části: `výkonný` *(execution)* a `konstrukční model` *(construction model)*. Výkonný model specifikuje jak budou služby vykonávány. Implementační kód služby, který bude proveden pro uskutečnění služby se nazývá `metoda`. Spuštění metody se pak označuje termínem `aktivace metody`. Při aktivaci objektu se zajišťují rozličné úkony, které jsou pro danou metodu specifické, jako je například perzistence. Opakem aktivace je `deaktivace metody`. Konstrukční model popisuje objektový stav, definice metod a různé politiky chování objektu.

Dále rozlišujeme dva druhy částí aplikace. První skupinu tvoří `klient`. Tato část přistupuje k distribuovanému objektu pomocí reference na objekt. Klient zná pouze logickou strukturu objektu reprezentovanou jejím rozhraním. Druhou skupinu tvoří `implementace objektu`. Někdy ji budu označovat termínem `server`. Tato skupina poskytuje objektu aplikační logiku *(business logic)*. Objektová implementace může mít řadu podob dosažených pomocí objektových adaptérů.

Někdy nelze absolutně označit části aplikace termínem server nebo klient. Označení se dá provést pouze relativně vždy k nějakému objektu, protože jeden objekt může být serverem k jednomu objektu, ale on sám může být klientem jiného.

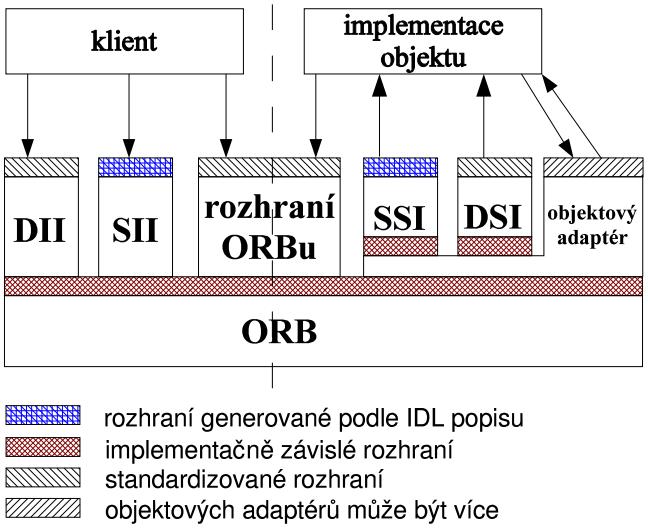
4.2 IDL

Pro popis objektového rozhraní, použitého při vývoji aplikací založených na technologii CORBA, slouží jazyk IDL (Interface Description Language). Popis vytvořený tímto jazykem slouží pak pro vytvoření kostry[1] programu závislé na konkrétním programovacím jazyku. V praxi to znamená, že popíšeme všechny distribuované objekty většinou pomocí tzv. CASE nástrojů[2], ve kterých se využívají technologie pro objektové modelování, jako je modelovací jazyk UML. Tyto nástroje potom vygenerují z daného objektového návrhu, v případě technologie CORBA, popis v jazyku IDL. Ne vždy jsou ovšem k dispozici drahé CASE nástroje, a proto je vhodné pečlivě nastudovat jazyk IDL, bez kterého se téměř žádná funkční aplikace vytvořit nedá. Vytvoření již zmiňované kostry programu probíhá za použití generátorů z jazyka IDL do konkrétního jazyka podporovaným ORBem. Generátory se dodávají většinou s celou implementací ORBu. Zde se začíná objevovat výhoda použití této techniky. Je patrné, že tak dosáhneme přenositelnosti návrhu aplikace mezi konkrétními programovacími jazyky a různými implementacemi architektury CORBA.

[1] kostra — stub a skeleton
[2] CASE — Computer Aided System Engeneering

4.3 ORB

Základní část celé technologie je postavena na ústřední komponentě ORB *(Object Request Broker)*. Tato komponenta je vlastně jakousi softwarovou sběrnicí. Je nízkoúrovňová a teprve na ní jsou stavěny další části technologie CORBA. Jejím základním úkolem je navázání spojení mezi komponenty aplikace a zajištění předání požadavků na jejich rozhraní. Požadavek se skládá z adresy vzdáleného objektu, požadované operace a jejích parametrů. K ORBu je možno přistupovat přes rozhraní, z kterého je pouze malá část standardizovaná. To ovšem programátorům nemusí vadit, protože na tomto implementačně závislém rozhraní jsou postaveny jiné části vyšší úrovně, které již mají rozhraní standardizováno. Programátor většinou s ORBem přímo nekomunikuje, z větší části jsou využity již zmiňované vysokoúrovňové části implementace, viz obrázek 4.1 na straně 21. Další důležitou úlohou je zajištění převodu přenášených dat do podoby nezávislé na platformě. Řeší se tak problém endiánovosti a jiných závislých vlastností na daných platformách. Tento převod se nazývá `marshaling`, jeho opakem je `demarshaling`, tedy převod z obecného formátu na formát konkrétní platformy. Oba tyto procesy jsou vůči programátorovi transparentní.



Obrázek 4.1: CORBA — základní struktura

4.4 DII a DSI

Pro zajištění komunikace mezi serverem a klientem jsou připraveny k využití programátory dvě komponenty. Na straně klienta je to část `DII` *(Dynamic Invocation Interface)* a na straně serveru `DSI` *(Dynamic Skeleton Interface)*. Tyto komponenty jsou obsaženy povinně v každé implementaci. V případě `DII` návrh umožňuje dynamicky za běhu aplikace vyvolávat metody na rozhraní distribuovaného objektu, které nebylo známo v době překladu. U `DSI` je situace obdobná, jenomže tato část bude volání zpracovávat a operace, které bude moci vyvolat budou zjišťovány až za chodu aplikace. Na druhou stranu obecnost těchto komponent částečně porušuje transparentnost při vývoji, převážně pak `DSI` je pro svou složitost v praxi prakticky nepoužitelná. Tento dynamický model se převážně využívá pro přemostění jednotlivých distribuovaných objektových systémů.

4.5 SII a SSI

Další způsob pro zprostředkování komunikace mezi klientem a serverem zajišťují další dvě komponenty. Klient využívá `SII` *(Static Invocation Interface)* a server zpracovává požadavky pomocí `SSI` *(Static Skeleton Interface)*. Tento způsob volání a zpracování požadavků převládá při vývoji aplikací. Jednak je tento stav dán transparentností všech operací pro vývojáře a dále automatizací vzniku jednotlivých rozhraní pro objekty v konkrétním implementačním jazyce. Ty jsou popsány v jazyku IDL a následně po jejich přeložení vzniká kostra programu, tzv. `stub` a `skeleton`. `Stub` představuje jakýsi zástupný *(proxy)* objekt vložený na straně klienta, který je specifický pro každé rozhraní. Naopak `skeleton` je odpovědný za zpracování příchozích požadavků na straně serveru a následné volání správné implementace požadovaných metod. Tento druh objektu se někdy označuje termínem `dispatcher`. Tento způsob komunikace je možný tehdy, jestliže jsou známa rozhraní objektu v době překladu. Programátor může volit mezi dynamickým a statickým modelem dle libosti. Obě dvě možnosti jsou transparentní vzhledem k druhé straně aplikace.

4.6 objektové adaptéry

Mezi ORBem a skeletony je umístěna vrstva obsahující objektové adaptéry. Ty mají zaručit schopnost správy jednotlivých objektů, přičemž jich může existovat v jedné chvíli několik paralelně zapojených. Objektových adaptérů je definováno hned několik druhů:

1. basic object adapter — (BOA)
2. portable object adapter — (POA)
3. library object adapter
4. object orientated database

Nejznámější z nich jsou pouze první dva. Objektový adaptér je vystaven nad implementačně závislým rozhraním ORBu a je od něj odvozena i implementace distribuovaného objektu. Z tohoto důvodu je vhodné, aby bylo objektových adaptérů co nejméně, jinak bude omezena přenositelnost objektu. Většinou jsou navrženy tak, aby pokrývaly co největší požadavky na jednotlivé typy objektů. Adaptéry nabízejí zejména následující služby:

- vytváření a implementace referencí na objekty
- vyvolávání metod
- aktivace a deaktivace objektů
- mapování referencí na objekty k implementacím
- registrace implementací objektů

4.6.1 BOA

Objektový adaptér BOA byl jediným adaptérem definovaným do verze 2.1. Jeho hlavní funkcí byla aktivace objektů na základě definovaných politik obsažených v `implementačním úložišti` *(implementation repository)*. Podporováno je pět druhů aktivací.

1. sdílená *(shared)* — aktivace tohoto druhu je asi jedna z nejpoužívanějších. BOA aktivuje požadovaný server a ten je pak schopný obsloužit libovolný počet klientů.
2. nesdílená *(unshared)* — tento druh aktivace je podobný sdílenému serveru, avšak jeden nesdílený server může obsloužit pouze jednu instanci klienta. Z toho plyne jednoduchá rovnice: „pro N instancí klienta, bude zapotřebí N instancí serveru“.
3. perzistentní *(persistent)* — perzistentní aktivace serveru je ve své podstatě sdílený server. Navíc se ale stará o aktivaci sám uživatel.

4. pro jednotlivé metody *(per-method)* — tato aktivace je podobná nesdílenému serveru. Rozdílem je, že zde bude vytvořená nová instance serveru pro každou vyvolanou metodu.

5. knihovní *(library)* — knihovní aktivace využívá zavádění dynamických knihoven serverem na vyžádání klientem.

Další zajímavou vlastností objektového adaptéru BOA je možnost perzistence instancí objektů a jejich migrace mezi jednotlivými servery za běhu programu, aniž by byla narušena funkčnost přistupujícího klienta. Perzistence zde nemá stejný význam jako u perzistentní aktivace, ale v tomto smyslu se jedná o serializaci instance.

Základní objektový adaptér nebyl zcela jasně definován a tak docházelo k různému výkladu od jedné implementace k druhé, což ničilo interoperabilitu jednotlivých komponent od zůzných výrobců a tak byl definován nový objektový adaptér POA.

4.6.2 POA

Po nejasnostech, které nastaly při implementaci různých verzí objektového adaptéru BOA vydala organizace OMG specifikaci pro nový adaptér POA. Při jeho návrhu vznikly nové termíny a jiné pozměnily svůj význam.

- `server` a `klient` — na rozdíl od předchozího významu se server a klient chápe v podobě celého procesu. Stále musíme mít na vědomí, že takto lze vzájemně pouze relativně označit procesy.
- `objekt` — abstraktní entita popisující CORBA objekt, tzn. entita s vlastní identitou zapouzdřující rozhraní a implementaci.
- `servant` — implementace objektu v konkrétním programovacím jazyce.
- `object id` — hodnota vyjadřující jednoznačné přiřazení mezi objektem a servantem. Přiděluje se při aktivaci objektu.
- `reference na objekt` — u ní navíc k původním uchovávaným hodnotám přibývají ještě `object id` a konkrétní identita adaptéru POA.

Vlastní adaptér POA je umístěný v kontextu serveru. V něm může být vytvářená hierarchická struktura obsahující další adaptéry POA, přičemž kořenový adaptér se označuje slovem `root`. Existence kořenového prvku je nutná pro inicializaci komunikace. Pro zpracování příchozích požadavků, ve smyslu nalezení odpovídajícího servanta, na objekt slouží tři způsoby, přičemž se mohou navzájem kombinovat.

1. Je udržována `mapa aktivních objektů`, ve které jsou uloženy informace o všech aktivovaných objektech v podobě relace mezi hodnotou `object id` a servantem. Objekt nebo servant se mohou nazvat aktivovanými, právě když jsou zaznamenány v mapě.
2. Používá se standardní servant, který zpracuje všechny požadavky na dosud neaktivované objekty.
3. Pro hledání vhodného servantu může být využit uživatelem napsaný manažer servantů. Při přítomnosti mapy aktivních objektů je v ní zaznamenán manažerem nalezený servant a hodnota `object id`. Pak již není nutné servant nikdy hledat.

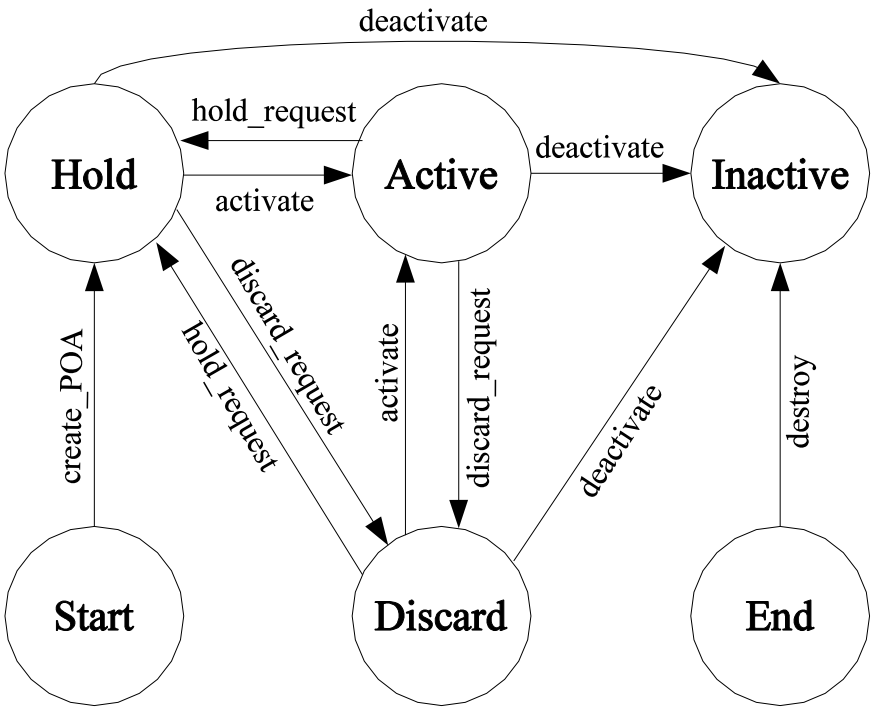
Tyto tři vlastnosti spolu s dalšími se dají nastavit pomocí politik adaptéru POA. Každý adaptér umístěný v hierarchické struktuře se může chovat podle své politiky. Kořenový POA `root` má standardně předdefinována následující pravidla.

- `ORB_CRT_MODEL` — doručení požadavků je uskutečněno na základě rozhodnutí ORBu. Při použití vícevláknového prostředí jich může být doručeno několik najednou.
- `TRANSIENT` — aktivované objekty nemohou svou životností přesáhnout dobu životnosti hostitelského procesu.
- `UNIQUE_ID` — servant může být aktivován pouze jednou, nemůže tudíž obsloužit více požadavků různých referencí na objekt.
- `SYSTEM_ID` — honota `object id` je přiřazena automaticky při aktivaci a je zaručena její jedinečnost v rámci jednoho adaptéru `POA`.
- `RETAIN` — adaptér bude udržovat mapu aktivních objektů
- `USE_ACTIVE_OBJECT_MAP_ONLY` — pro hledání odpovídajícího servantu bude využita pouze mapa aktivních objektů, přičemž pokud nebude odpovídající servant aktivní, bude vyhozena výjimka.
- `NO_IMPLICIT_ACTIVATION` — vyhození výjímky při použití neaktivního servanta v kontextu aplikace, kde je očekáván aktivní.

Politik je k dispozici daleko více než používá kořenový adaptér. Pro změnu politik se musí vytvořit nová instance adaptéru POA, kde při jejím vzniku je jí přiřazen nový soubor politik, který je po dobu její existence neměnný.

Dalším vylepšením oproti adaptéru BOA je POA manažer. Pomocí něho lze snadno dosáhnout úplné kontroly nad příchozími požadavky. Samotný je implementován jako

objekt. Jeho instance jsou poté automaticky přiděleny jednotlivým adaptérům POA. Manažer je vyobrazen na obrázku 4.2, přičemž kruhy označují jednotlivé stavy a šipky označují operace potřebné vykonat pro přechod do dalších stavů.



Obrázek 4.2: POA manažer

Hold — příchozí požadavky jsou řazeny do fronty. Po jejím naplnění je vyvolána výjimka `TRANSIENT` a další příchozí požadavky jsou zahazovány. V tomto stavu se nachází každý manažer při svém vzniku.

Active — příchozí požadavky jsou zpracovány okamžitě. Pokud je přijato více požadavků. než je možno v jednom okamžiku zpracovat, jsou posléze řazeny do fronty. Velikost fronty není standardizována. Při jejím naplnění je opět vyhozena výjimka `TRANSIENT`.

Discard — příchozí požadavky jsou zahozeny a je vyvolána výjimka `TRANSIENT`. Tento stav se většinou využívá při zahlcení serveru požadavky.

Inactive — poslední stav před destruováním, zde požadavky nejsou již nadále přijímany. Tento stav je nezvratný. Klient vždy obdrží výjimku `OBJ_ADAPTER`.

4.7 interface a implementation repository

Specifikace předepisuje implementaci dvou služeb, které uchovávají různé informace o sdíleném objektu. Tyto služby jsou většinou dodávány v podobě samostatných aplikací. Jednak se ukládají v `interface repository` informace o rozhraní a do `implementation repository` se vkládají informace týkající se vlastní implementace objektu. K úložišti rozhraní se většinou přistupuje při dynamickém sestavování požadavku za běhu *(DII)* anebo při modelování aplikace pomocí CASE nástrojů. Rozhraní pro přístup k uloženým rozhraním je přesně stanoven. Zatímco rozhraní implementačního úložiště je závislé pouze na kreativitě dodavatele. V tomto úložišti jsou většinou zaznamenány informace o aktivaci objektů, jejich zabezpečení a fyzickém umístění.

4.8 komunikace

Organizace OMG stanovila pro svoji technologii obecný protokol `GIOP` *(General Inter-ORB Protocol)* pro přenos dat. Jeho výhodou je, že implementace každého ORBu musí znát pouze jeden protokol, aniž by se musela zabývat síťovou strukturou. Nad obecným formátem se následně budují další protokoly, které již úzce spolupracují s konkrétním transportním protokolem. Vzhledem k tomu, že v současnosti jsou zřejmě nejrozšířenější sítě postavené na bázi rodiny protokolů TCP/IP, je od verze 2.1 definován protokol `IIOP`. Pro komunikaci mezi různými standardními i nestandardními nadstavbami `GIOP` se využívají tzv. mosty. Mohou se využít pro přemostění různých technologií distribuovaných systémů. Data se přenášejí v obecném formátu `CDR` *(Common Data Representation)*.

Protokol GIOP je kompaktní a o tom svědčí typy zpráv, které mohou být posílány mezi serverem a klientem.

Klient může vyslat následující zprávy:

- Request — vyvolání metody vzdáleného objektu
- Cancel Request — zrušení předcházejícího požadavku
- Locate Request — zjištění umístění vzdáleného objektu
- Message Error — reakce na předchozí zprávu

Server může zaslat následující typy zpráv:

- Reply — výsledek vyvolání

- Locate Reply — indikuje, zda server implementuje objekt, nebo předá volání dále
- Close Connection — uzavření spojení
- Message Error — reakce na předchozí zprávu

4.9 COSS

V předmluvě k této práci byla zmíněna vlastnost znovupoužitelnosti kódu dosažené pomocí objektově orientovaného přístupu. Organizace OMG standardizovala několik objektových služeb pro technologii CORBA. V současné době by měly být na trhu k dispozici implementace všech níže jmenovaných služeb. Tento oddíl se snaží stručně charakterizovat některé ze standardizovaných objektových služeb. Služby, které se dle mého názoru využívají nejvíce, jsou popsány podrobněji a u zbylých služeb je uveden pouze stručný popis.

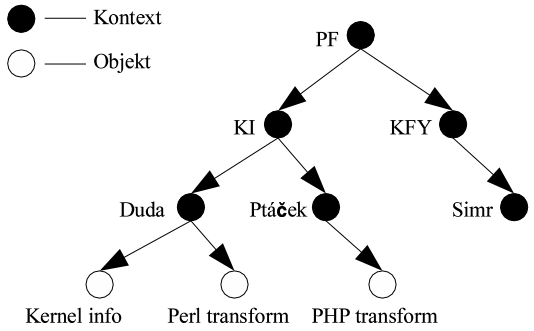
Všechny standardizované služby, podobně jako implementace ORBu, nemají pevně stanovené podmínky pro implementaci. Jediné, co musí být dodrženo, je definované rozhraní. Skupina OMG se při návrhu soustředila na jednoduchost rozhraní. Služby neobsahují žádné jiné vlastnosti než ty, pro které byly navrženy. Při takto jednoduchém návrhu se dají služby dle potřeby vzájemně kombinovat a lze tak dosáhnout komplexního zázemí pro tvorbu distribuovaných aplikací.

Naming service — jmenná služba je obdobou služby DNS[3], která se vyskytuje v počítačových sítích využívající protokol IP. Zde se ovšem netvoří relace mezi IP adresou a doménovým jménem, ale udržuje se vztah mezi jménem objektu a referencí na objekt. Jeden objekt může mít přiřazeno více jmen. Jméno je složeno z identifikátoru a druhu objektu. Tyto dvě položky jmenná služba nezpracovává a tudíž má jejich obsah pouze informativní charakter. Takto vzniklou databázi si lze představit jako strom, viz obrázek 4.3. Všechny uzly stromu jsou objekty. Listy stromu, až na výjímky, jsou objekty uchovávající uspořádanou dvojici jména a reference objektu. Zbylé uzly vytvářejí tzv. kontext. Jediný případ, kdy list stromu reprezentuje kontextový objekt nastává tehdy, pokud se v daném kontextu nenacházejí žádné objekty. Jméno musí být jedinečné v daném kontextu. Tento strom nemusí mít pevně definovaný kořen a proto nelze stanovit absolutní cestu. Určitý objekt se může vyjádřit pouze ve vztahu k nějakému kontextu.

Jelikož nejsou stanoveny podmínky implementace, lze touto službou zajistit zapouzdření již existující adresářové technologie jako je například LDAP. Pro organizace využívající síť postavenou na operačním systému Novell Netware může

[3]DNS — Domain Naming Service

být výhodné využít adresářovou službu NDS a zapouzdřit jí jmennou službou. Vznikne tak velmi silná databáze prostředků sítě a usnadní se tak správa. Zapouzdřenost může být i víceúrovňová. Návrh této služby se zdá být poměrně flexibilní a v praxi se také často využívá.



Obrázek 4.3: Naming service — stromový graf

Event service — služba pro šíření událostí implementuje podporu pro asynchronní přenos informací mezi objekty. Objekty, které tuto službu využívají, nemusí navzájem o sobě znát mnoho informací. Rozlišují se dva druhy objektů: producenti a konzumenti. Producenti vysílají data a konzumenti je zpracovávají. Při komunikaci se používají dva modely. První je model *push*. V rámci modelu *push* zahajuje přenos události producent. V druhém modelu *pop* konzument vyšle žádost o událost a producent na tuto žádost odpoví vysláním události. Veškerá komunikace se odehrává přes tzv. kanál událostí. Tento kanál je reprezentován objektem, který je těsně spjat s ORBem. Existuje generický kanál a typový kanál. Jednodušší je použití generického kanálu. Při takové komunikaci nese událost informaci obsaženou v jednom parametru, který je typu `any`. V tomto případě nemusí být známo nic o obsahu přenášené informace. Zcela odlišný je typový kanál. Typ přenášených informací zde hraje svoji roli. Pomocí IDL lze definovat typy parametrů, přičemž všechny parametry musí být vstupní. Na základě typů lze potom události filtrovat, takže se sníží počet nevyžádaných událostí přijatých ze strany konzumenta.

Komunikační kanál je možné při předávání událostí vynechat, vznikají tak přímé nezprostředkované události *(Point-to-Point events)*. Aby byla komunikace možná, musí si nejprve konzument s producentem vyměnit referenci na `PushConsumer` a `PushSupplier` objekt. Pro doručení události ke konzumentovi pak producent volá na referenci objektu `PushConsumer` metodu `push`.

Přímé nezprostředkované události nejsou příliš transparentní a objekty se musí vypořádat s různými problémy, jako jsou nedoručitelnost události nebo uchování všech spojení. Následkem toho se většinou tento komunikační typ nevyužívá.

Transaction service — transakční služba je zřejmě jednou z nejdůležitějších a nejpoužívanějších služeb, protože zajišťuje spolehlivost distribuovaných aplikací. S pomocí této služby se mohou definovat transakce. Ty jsou základní jednotkou obnovy a konzistentnosti v distribuovaných systémech. Pro transakce lze využít pouze objekty, které je podporují. V modelu CORBA je toho dosaženo odvozením objektů od obecných tříd obsažených v transakční službě. Existuje mnoho transakčních modelů, ale podle specifikace je vyžadován pouze základní *(flat)* model a dále služba může používat vnořený *(nested)* model. Transakční model se rozlišuje podle následujících aspektů:

1. začátek transakce
2. konec transakce
3. jednotka obnovy při selhání

Transakci vždy začne vykonávat klient, následuje zpracování požadavků na libovolném počtu serverů a skončí opět u zahajujícího klienta. V případě základního modelu je vše vykonáno lineárně. Tento model lze charakterizovat slovy *„všechno nebo nic“*. Transakce může být zakončena pouze jejím selháním nebo potvrzením. Vnořený model přináší oproti svému předchůdci větší míru pružnosti. Uvnitř jedné transakce lze definovat více podtransakcí. Vrchní transakce je vždy odvozena od základního modelu a zahajuje volání podtransakcí. Vzniká tak hierarchické uspořádání, které se prochází rekurzivním voláním. Podtransakce bude nezvratná pokud všichni její předchůdci byly ukončeny potvrzením o úspěchu a zároveň aktuální transakce skončila s úspěchem. Pokud aktuální transakce bude zakončena neúspěchem, všichni její potomci budou ukončeny rovněž s neúspěchem. Oba dva modely lze použít i v heterogenním prostředí s různými implementaci ORBu.

S touto službou se často využívá služba řízení konkurenčního přístupu.

Concurrency control service — služba řízení konkurenčního přístupu zajišťuje přístup ke sdíleným prostředkům tak, aby nebyla porušena jejich konzistentnost. Tato služba byla navržena pro použití v transakčním i normálním režimu zpracování. Řízení přístupu je realizováno pomocí zámků, přičemž se nerozlišuje zda byl použit v transakci či nikoli. To umožňuje sériový přístup k prostředku aniž by nastávaly konflikty při různých režimech zpracování. Rozlišuje se pět druhů zámků.

1. čtecí
2. zapisovací
3. záměrný čtecí *(intention read)*
4. záměrný zapisovací *(intention write)*
5. upgrade

Externalization service — externalizační služba slouží k uložení stavu objektu do proudu dat. Proud dat můžeme potom kdekoliv a kdykoliv použít v jiném procesu. Tím může být docíleno transparentního přesunu objektů.

Persistent object service — perzistentní objektová služba může být pověřena správou stavů jednotlivých objektů, přičemž data jsou uložena nejčastěji v jednotlivých souborech, objektových nebo relačních databázích.

Query service — dotazovací služba je používána k hledání množiny objektů podle určených kritérií. Při dotazování nedochází k porušení zapouzdřenosti jednotlivých objektů. Pro sestavování dotazu je využíván jazyk OQL *(Object Query Language)*, jenž je standardizovaný organizací SQL3 ANSI X3H2. Podmnožinou OQL je i celá část dotazovacího jazyka SQL-92. Dotazovací služba je většinou použita ve spojení perzistentní objektovou službou.

Relationship service — smyslem relační služby je tvorba vzájemných vztahů mezi objekty, které o sobě nemusí vzájemně mít žádné informace. Relace jsou tvořeny až za běhu programu. Existuje jich několik typů, přičemž nejdůležitější z nich je relace vlastnictví.

Time service — časová služba slouží k synchronizaci časových údajů jednotlivých počítačů. Kromě synchronizace ji lze použít k zjištění intervalu během dvou událostí a ke generování událostí založených na časovačích.

Licensing service — licenční služba slouží k poskytování licencí jednotlivých objektů na základě začátku, trvání nebo konce její platnosti. Dle publikace [DOB-05] je tato služba tou poslední, která by byla implementována vývojáři open source softwaru.

Property service — pomocí této služby lze dosáhnout dynamického vytvoření atributů objektu během jeho životnosti. Lze tak prolomit bariéru „statického“ jazyka IDL. Vytvořené atributy mají své jméno, hodnotu a přístupové režimy.

Security service — bezpečnostní služba je jednou z nejrozsáhlejších služeb, které kdy byly definovány. Vychází z předpokladu, že cílový objekt nemusí naprosto nic vědět o implementaci bezpečnosti. Tím se bezpečnost znatelně zvýší. Podporována

je identifikace a autentifikace klienta, autorizace a řízení přístupu, bezpečnostní audit, bezpečnost komunikace a nepopiratelnost převzetí.

Life cycle service — služba životního cyklu je určena pro přesouvání, kopírování a destruování množin objektů. Většinou je používána spolu s relační službou. Její hlavní předností hluboká *(deep)* manipulace s objekty, tzn. manipuluje s objekty, které jsou vlastněny daným objektem.

Trading object service — tato služba má na starosti nalezení objektů na základě služeb, které poskytují. Spolu s jmennou službou se pak dají lokalizovat jakékoli objekty.

Kapitola 5

Rozhraní

5.1 OMG IDL — C++ mapování

Jazyk IDL má mnoho dialektů a model CORBA využívá právě dialekt OMG. Dále v textu bude uváděno pouze označení IDL, ale samozřejmě se bude jednat o dialekt OMG. Pro popisný jazyk IDL definovala a spravuje organizace OMG mapování pro nejvíce používané programovací jazyky. Pro převod do konkrétního jazyka slouží generátory dodávané spolu s implementací ORBu. Jazyk IDL se podobá svou syntaxí jazyku C++. Specifikace dovoluje použití preprocesoru jazyka C++ pro zpracování jazyka IDL, proto je dobré si nejprve ověřit, zvláště na uzavřených systémech, zda generátor je plnohodnotný. Následující část této práce popisuje mapování do jazyka C++. Pro popis syntaxe je využita zjednodušená Backus–Naurova forma zápisu.

5.1.1 obecná pravidla

Jazyk IDL rozlišuje velká a malá písmena, nelze ovšem na jejich základě činit definice stejného názvu pomocí velikosti písmen ve stejné oblasti viditelnosti. Všechny definice jsou ukončeny znakem středník, rovněž tak je i ukončen blok definic. Rovněž tak jako v jazyce C++ je i v OMG IDL stejné značení pro komentáře. Rozlišují se dva typy:

1. celořádkový komentář — začíná znaky `//`
   ```
   // zde je celořádková poznámka IDL
   ```
2. blokový komentář — začíná znaky `/*` a končí `*/`
   ```
   /* zde je bloková poznámka IDL */
   ```

Parametry všech metod musí mít uvedena jména identifikátorů. Zároveň nelze uvést deklaraci metody, aniž by neměla specifikovaný návratový typ.

Použití klíčového slova C++ jako názvu identifikátoru v některé z deklaracích IDL bude mít za následek mapování na identifikátor se stejným jménem jako klíčové slovo s prefixem `_cxx_`. Tím se zabrání konfliktům. Veškeré podpůrné objekty budou mít také uveden prefix.

5.1.2 základní typy

- **`short`** — celočíselný 16bitový datový typ s rozsahem „$-2^{15} \cdots 2^{15} - 1$“ mapovaný na typ `CORBA::Short`
- **`long`** — celočíselný 32bitový datový typ s rozsahem „$-2^{31} \cdots 2^{31} - 1$“ mapovaný na typ `CORBA::Long`
- **`long long`** — celočíselný 64bitový datový typ s rozsahem „$-2^{63} \cdots 2^{63} - 1$“, jenž se mapuje na typ `CORBA::LongLong`
- **`unsigned short`** — celočíselný nezáporný 16bitový datový typ s číselným rozsahem „$0 \cdots 2^{16} - 1$“ mapovaný na typ `CORBA::UShort`
- **`unsigned long`** — celočíselný nezáporný 32bitový datový typ mapovaný na typ `CORBA::ULong` s rozsahem „$0 \cdots 2^{32} - 1$“
- **`unsigned long long`** — celočíselný nezáporný 64bitový datový typ mající rozsah „$0 \cdots 2^{64} - 1$“, jenž je mapován na typ `CORBA::ULongLong`
- **`float`** — reálný 32bitový datový typ s jednoduchou přesností mapovaný na typ `CORBA::Float`
- **`double`** — reálný 64bitový datový typ s dvojitou přesností mapovaný na typ `CORBA::Double`
- **`long double`** — reálný minimálně 80bitový datový typ mapovaný na typ `CORBA::LongDouble`
- **`char`** — 8bitový celočíselný datový typ s konverzí při přenosu mapovaný na typ `CORBA::Char`
- **`wchar`** — celočíselný datový typ mapovaný na typ `CORBA::WChar`, jeho velikost je dána implementací, při použití v distribuovaném prostředí probíhá konverze
- **`boolean`** — 8bitový celočíselný datový typ mapovaný na typ `CORBA::Boolean`, jsou definovány pouze hodnoty 0 a 1, ostatní stavy nejsou definovány

- **octet** — 8bitový celočíselný datový typ mapovaný na typ `CORBA::Octet`, při přenosu není použita žádná konverze
- **void** — typ pro metody, které nevrací hodnotu.

Všechny základní typy musí mít podle specifikace zajištěnou odpovídající reprezentaci typů v konkrétním programovacím jazyku. Reprezentace je prováděna dvěma způsoby. První způsob využívá již existujících datových typů v konkrétním jazyku a může být proveden například konstrukcí v jazyce C++ [1]`typedef skutecny_typ CORBA::typ;`. Druhý způsob reprezentace je zajištěn pomocí vlastních typů, které implementace bude vnitřně používat. Pro programovací jazyk C++ může implementace využít vnitřně vestavěné typy. Z výše uvedených typů nemusejí být mapovány na nativní typy C++ pouze `boolean`, `char`, `wchar` a `octet`. Při pohledu na předchozí seznam si lze povšimnout, že všechny typy mají stanovenou pevnou velikost. Nemělo by se využívat pro mapování do jazyka C++ základního typu `int`. Tento typ by neměl být použit, protože má velikost datového registru procesoru. Vzhledem k tomu, že technologie CORBA je multiplatformní, nelze zajistit odpovídající reprezentaci typu `int` na různých hardwarových platformách. To znamená, že implementace by nebyla přenositelná pro různé typy hardwaru, i kdyby byl použit stejný operační systém.

5.1.3 složené typy

Podobně jako v C++ existují i v IDL prakticky stejné složené typy: výčtový typ `enum`, strukturované typy `struct`, `union` a třídy objektů `interface`. Dále lze za složený typ v jazyce IDL považovat řetězce `string`, `wstring`.

- **enum** — výčtový typ IDL je mapován do jazyka C++ na jeho vlastní výčtový typ `enum`. Není zde zaručeno automatické číslování položek. Počet položek je minimálně roven počtu položek definovaných v jazyce IDL. Vždy musí být výčtový typ zarovnán na 32 bitů. Toto zarovnání zajistí vždy generátor, uživatel tudíž nemusí znát velikost použitých typů.

```
enum identifikátor
{
    identifikátor_prvku [, identifikátor_prvku ]...
};
```

Výpis 5.1: Syntaxe výčtového typu enum

[1] příklad: `typedef bool CORBA::Boolean`

- **struct** — strukturovaný typ IDL je dle standardu mapován na strukturovaný typ `struct` jazyka C++. Položky obsažené ve struktuře IDL jsou mapovány na typy jazyka C++ podle pravidel uvedených ve specifikaci. Struktury jsou vždy předávané hodnotou a nikoli referencí, eliminuje se tak „*callback*[2]“ efekt. Při použití přiřazovacího operátoru mezi instancemi typu `struct` nebo při plnění položek struktury se provádí hluboké kopírování[3]. Struktura se tak stává vlastníkem obsahu položek a je odpovědná za jejich uvolnění.

```
struct identifikátor
{
    typ_položky identifikátor_položky;
    [typ_položky identifikátor_položky;]...
};
```

Výpis 5.2: Syntaxe strukturovaného typu struct

- **union** — strukturovaný typ vytvořený pomocí IDL konstruktu `union` kombinuje vlastnosti konstruktů `union` a `switch` jazyka C++. Konstrukt IDL `union` je podobný struktuře až na to, že při použití tohoto složeného typu jako parametr metody se předávají pouze inicializované položky. Konstrukt `switch` je zde použit proto, aby bylo možné zjistit, které z položek byly inicializovány. Pořadí inicializovaných položek dané hodnotou v konstrukci `switch` se nazývá diskriminátor a může být celočíselného nebo výčtového typu. Podobně jako u struktury se při použití přiřazovacího operátoru mezi instancemi typu `union` nebo při plnění položky unionu provádí hluboké kopírování. Uniony jsou vlastníky obsahu položek a jsou odpovědny za jejich uvolnění. Dříve byly uniony využívány pro snížení zátěže v počítačové síti. V praxi se uniony téměř v dnešní době nepoužívají, jelikož sítě již mají velikou propustnost.

```
union identifikátor
switch (typ_diskriminátoru)
{
    [case konstantní_hodnota:
        [typ_položky identifikátor_položky;]...
    ]...
    [default: typ_položky identifikátor_položky;]
};
```

Výpis 5.3: Syntaxe strukturovaného typu union

[2]callback efekt — vzájemné volání funkcí
[3]hluboké kopírování (deep copy) — duplikace celých objektů, nevytváří se pouze kopie jejich adres

Konstrukce `switch` musí obsahovat alespoň jednu položku. Pokud větev `CASE`, která odpovídá hodnotě diskriminátoru, obsahuje více položek, předpokládá se, že tyto položky budou všechny inicializovány. Avšak pokud inicializovány nebudou, nastane při použití nedefinovaný stav. Specifikace nevynucuje, aby implementace zjišťovala a případně upozorňovala na tuto programátorskou chybu. Typ IDL `union` je mapován v jazyce C++ na třídu, která obsahuje přístupové metody ke všem definovaným položkám a k diskriminátoru.

- **`interface`** — tento konstrukt má na starosti vytvoření rozhraní distribuovaného objektu. Jelikož jazyk C++ neobsahuje patřičnou konstrukci, je tento typ mapován na třídu, na rozdíl od jazyka Java, který má konstrukci `interface` s ekvivalentním významem. Rozhraní může obsahovat libovolný počet metod a definic.

```
interface identifikátor [: předek][, předek]
{
    [atribut;]...
    [metoda;]...
};
```

Výpis 5.4: Syntaxe strukturovaného typu interface

Program využívají distribuované objekty nesmí nikdy vytvářet nebo jakýmkoli způsobem vlastnit instance rozhraní. Dále nesmí používat ukazatele nebo reference na rozhraní. Zákaz tohoto druhu ponechává větší volnost při implementaci objektů.

dědičnost

Jazyk IDL umožňuje definovat dědičnost pro rozhraní. Pomocí ní můžeme specializovat rozhraní, přičemž může být využita i vícenásobná dědičnost. Implementace rozhraní potom vypadá následovně. Po vytvoření `skeletonu` vznikne třída identifikovatelná pomocí názvu, který je tvořen z identifikátoru rozhraní a postfixu `_skel` v případě adaptéru BOA, nebo prefixu `POA_` při použití adaptéru POA. Od `skeletonu` se odvodí implementace rozhraní, viz strana 52. Pokud rozhraní používá vícenásobnou dědičnost, nabízejí se dvě možnosti.

1. Implementace odvozené třídy bude dědit od implementace obecného rozhraní a skeletonu odvozené třídy. Přičemž musíme zajistit, aby byl konstruktor skeletonu volán jako poslední.
2. Implementace odvozené třídy bude dědit od skeletonu obecné třídy a skeletonu odvozené třídy.

Všechny implementace objektů jsou odvozeny od nejobecnějšího rozhraní `Object`.

metody

Metody popsané IDL jazykem mají jednoduchou syntaxi. Při psaní deklarací se neuvádějí žádné konstruktory ani destruktory, protože by to bylo z důvodu následné implementace objektu bezvýznamné. Všechny metody jsou mapovány jako veřejné.

```
[oneway] typ identifikátor ([parametr[, parametr]...])
[raises (výjimka[, výjimka]...)];
```

Výpis 5.5: Syntaxe deklarace metod rozhraní

Modifikátor `oneway` označuje metodu, jež bude neblokující. Tím pádem musí být návratová hodnota typu `void` a všechny předávané parametry musí sloužit pouze pro vstup. Při neúspěchu nebude vyvolána žádná výjimka a tudíž nelze zaručit notifikaci o úspěchu.

Parametr se skládá z modifikátoru směru, typu a identifikátoru. Modifikátor směru muže nabývat tří hodnot: vstupní `in`, vstupně-výstupní `inout` a výstupní `out`. V závislosti na něm jsou prováděny konverzní operace. Pro vstupní a vstupně-výstupní směr se provádí marshaling při odesílání požadavku na volání metody a pak při návratu z ní se znovu provádí marshaling pro argumenty výstupního a vstupně-výstupního směru. Dále bude vykonán marshaling na návratovou hodnotu.

Novější specifikace verze 2.4 již s sebou přináší možnost asynchronního volání metod.

Programátor musí dodržovat jisté konvence pro alokování a uvolňování paměti. Ty jsou zobrazeny v tabulce 5.1 na straně 39, kde je ke každému typu přiřazeno číslo skupiny pravidel.

1. Klient musí alokovat paměť podle použitých datových typů. Vstupně-výstupní proměnné musí inicializovat. Výstupní proměnné nemusí být inicializovány, protože stejně se na nich nebude provádět při volání marshaling.

2. Klient naalokuje paměť pro referenci na objekt. Pro vstupně-výstupní parametry zajistí inicializaci. Server musí následně v případě změny argumentu paměť nejprve uvolnit. Aby klient mohl používat dále stejnou referenci, musí ji zduplikovat. Klient je po návratu z volané funkce odpovědný za uvolnění všech výstupních parametrů včetně návratové hodnoty.

3. Pro výstupní hodnoty musí klient alokovat ukazatel na konkrétní typ a předat ho referencí. Server pak naalokuje paměť a nikdy nemůže vrátit nulovou hodnotu. Vrácená data nesmí být klientem modifikována. O uvolnění paměti se postará klient.

typ	in	inout	out	návratová hodnota
`short`	1	1	1	1
`long`	1	1	1	1
`long long`	1	1	1	1
`unsigned short`	1	1	1	1
`unsigned long`	1	1	1	1
`unsigned long long`	1	1	1	1
`float`	1	1	1	1
`double`	1	1	1	1
`long double`	1	1	1	1
`boolean`	1	1	1	1
`char`	1	1	1	1
`wchar`	1	1	1	1
`octet`	1	1	1	1
`enum`	1	1	1	1
`unsigned short`	1	1	1	1
reference na objekt (ptr)	2	2	2	2
`struct` (pevná délka)	1	1	1	1
`struct` (proměnlivá délka)	1	1	3	3
`union` (pevná délka)	1	1	1	1
`union` (proměnlivá délka)	1	1	3	3
`string`	1	4	3	3
`wstring`	1	4	3	3
`sequence`	1	5	3	3
`array` (pevná délka)	1	1	1	6
`array` (proměnlivá délka)	1	1	6	6
`any`	1	5	3	3
`fixed`	1	1	1	1

Tabulka 5.1: Alokační konvence pro parametry metod

4. Pro vstupně-výstupní řetězce alokuje a inicializuje klient paměť. To by mělo být prováděno pomocí funkcí pro řetězce standardizovaných v technologii CORBA. Eliminuje se tím problém přetečení dat při návratu ze serveru. O uvolnění se opět stará klient.
5. Zde musí klient dbát na to, aby nepoužíval data ve vlastnictví vstupně-výstupních parametrů po volání operace, protože mohou být kdykoli uvolněna.
6. Pro výstupní argumenty typu `array` musí klient alokovat ukazatel na paměť o stejných dimenzích jako má originál až na první. Poté co ho předá referencí server, tak bude serverem nastaven na platnou instanci pole. Nikdy nesmí zůstat nastaven na nulu. Klient nesmí nikdy měnit hodnoty vráceného pole a je povinen ho po skončení používání uvolnit.

atributy

Rozhraní může dále obsahovat atributy. Mohou být libovolného typu a každý atribut má při překladu IDL vygenerovány dvě metody, jednu pro čtení a druhou pro zápis dat z atributu. Pro zákaz zápisu do atributu, kromě členských metod objektu, lze definovat modifikátor `readonly`. Potom je vygenerována pouze čtecí metoda.

```
[readonly] attribute typ identifikátor [, identifikátor]...;
```

Výpis 5.6: Syntaxe atributu rozhraní

5.1.4 výjimky

Architektura CORBA má velmi dobře propracovaný systém výjimek. Při chybě volané metody je vyvolána výjimka, provádění metody se okamžitě přeruší a pokud byla specifikována výjimka pro právě nastolený neočekávaný stav, tak bude distribuována do volající metody. Ta ji může ošetřit nebo ji nechat šířit dál. Pokud by nebyla zachycena ani funkcí `main`, bude celá aplikace ukončena.

Výjimky dělíme na dva druhy: definované uživatelem a systémové. Pro jejich zachycení je používán standardní mechanizmus jazyka C++ „try/catch“.

```
exception jméno_výjimky {
    typ název_členu;
    [typ název_členu;]...
};
```

Výpis 5.7: Syntaxe výjimky

5.1.5 ostatní typy a konstrukce

typ any — Poslední typ, o kterém se zde zmíním, je typ `any`. Mapuje se do jmenného prostoru `CORBA` na typ `Any`. Implementuje se jako třída. Tento typ zaujímá mezi ostatními zvláštní postavení. Slouží jako zástupce pro ostatní IDL typy, ba dokonce i pro ty, které jsou neznámé během překladu. Pro lepší pochopení funkce tohoto mechanizmu je vhodné uvést náznak implementace.

Vzhledem k tomu, že `any`, může zapouzdřit libovolný typ, je nezbytné mít k dispozici informaci o uchovávaných datech. K tomu slouží pseudotyp `TypeCode`. Konstanty `TypeCode` jsou vygenerovány po překladu IDL deklarací. Zajištění vytvoření konstanty pro neznámé typy za běhu se provádí dvěma způsoby. Buď je získána z `Interface Repository` nebo pomocí metod nabízených ORBem.

Pro práci s daty jsou v C++ přetíženy operátory <<= a >>=. Ty jsou dostatečné pro většinu používaných typů, avšak i zde existují výjimky. Pro vstup dat slouží operátor <<= a ten je přetížen pro každý typ následujícími pravidly:

1. `void operator<<= (CORBA::Any&, typ);`
2. `void operator<<= (CORBA::Any&, const typ&);`
3. `void operator<<= (CORBA::Any&, typ*);`

První pravidlo je definováno pro typy, které jsou předávány hodnotou. Jmenovitě se jedná o následující typy:

- základní typy `short`, `long`, `long double`, `unsigned short`, `double`, `float`, `unsigned long long`, `unsigned long` a `long long`
- výčtové typy `enum`
- řetězce předávané hodnotou `string` a `wstring`
- objektové reference
- ukazatelé na typ `valuetype`[4]

Poslední dvě pravidla jsou použita pro typy, které jsou větších velikostí a předávání hodnotou by bylo výpočetně náročné. První z nich je kopírovací verze a druhé je nekopírovací. Jedná se o následující typy:

- strukturované typy `struct` a `union`
- kontejnerové typy

[4] `valuetype` — speciální konstrukce implementující určitá rozhraní, může být předána hodnotou

- výjimky
- typ any

Jak je výše vidět, typ any může být rekurzivně vnořen sám do sebe.

Pro extrahování dat zpět z typu any slouží operátor >>=. Opět je přetížen pro každý typ podle následujících pravidel:

1. `CORBA::Boolean operator>>= (const CORBA::Any&, typ&);`
2. `CORBA::Boolean operator>>= (const CORBA::Any&, const typ*&);`
3. `CORBA::Boolean operator>>= (const CORBA::Any&, typ*&);`

První pravidlo je kopírovací a je použito, opět jako při vkládání, pro základní typy, výčty apod.. Druhé pravidlo je nekopírovací a je použito k vyjmutí objemnějších dat z typu any *(viz vkládání dat)*. Přetížení dle posledního pravidla je nyní zastaralé až na jednu výjimku. Vždy bude využito při vyjmutí dat typu valuetype, protože volané operace při jeho používání neumějí pracovat s modifikátorem const.

Návratové hodnoty přetížených operátorů pro vyjmutí reprezentují úspěch TRUE nebo neúspěch celé operace FALSE. Nikdy nesmí nastat snaha o uvolnění paměti jakýmkoli způsobem. Za její uvolnění je zodpovědná implementace typu any.

Existují ale také IDL typy, které nejsou jednoznačně mapovány, a může nastat konflikt se základními typy. Pro ně jsou definovány statické metody, které zajistí správné přetypování objektu a přidělení jednoznačné konstanty TypeCode. Jedná se o následující typy uvedené v tabulce 5.2 (pro typy byl vynechán jmenný prostor CORBA).

Typ	Metoda pro uložení	Metoda pro vyjmutí
boolean	from_boolean(Boolean)	to_boolean(Boolean&)
octet	from_octet(Octet)	to_octet(octet&)
char	from_char(Char)	to_char(Char&)
string	from_string(char*,ULong)	to_string(char*&,ULong)

Tabulka 5.2: Metody pro správné určení typu v CORBA::Any

Při použití nekopírovací verze metody pro vkládání se již nadále nesmí přistupovat k originálu dat, protože typ any považuje jejich paměť za jeho vlastnictví a následně se stará o její uvolnění. Po volání kopírovací metody je vše transparentní

a k obsahu se může přistupovat i nadále. Při vyjmutí platí obdobné podmínky, pouze s tím rozdílem, že při použití kopírovací verze musí klient na konci používání uvolnit paměť vyextrahovaných dat.

konstrukce typedef — je určena pro přiřazení uživatelských identifikátorů základním a rozšířeným typům, které byly předem deklarovány.

```
typedef zastupovaný_typ alias [, alias];
```

Výpis 5.8: Syntaxe konstrukce typedef

konstrukce module — pomocí ní se vytvoří jmenný prostor, ve kterém budou umístěny všechny následné deklarace.

```
module jmenný_prostor
{
    [deklarace;]...
};
```

Výpis 5.9: Syntaxe konstrukce module

předběžné deklarace — jsou používány stejně tak jako v jiných programovacích jazycích, aby se vyřešil problém cyklické závislosti.

```
struct | interface | union identifikátor;
```

Výpis 5.10: Syntaxe předběžné deklarace

modifikátor const — pomocí tohoto modifikátoru se mohou vytvářet konstanty. Mohou být znakového, octet, řetězcového a výčtového typu. Při definování hodnoty konstanty lze použít běžné aritmetické operace včetně bitových posunů.

```
const typ jméno_konstanty = výraz;
```

Výpis 5.11: Syntaxe konstanty

kontejnerový typ — jazyk IDL podporuje jeden kontejnerový typ. Pomocí něj lze vytvořit dynamicky proměnné pole hodnot stejného typu. Jedná se o sekvenční typ. Za položku *typ* se doplní existující typ, z kterého bude sekvence sestavena. Pokud specifikujeme délku, sekvence může vlastnit maximálně počet položek roven délce.

```
typedef sequence<typ [, délka]> nový_typ;
```

Výpis 5.12: Syntaxe sekvenčního typu

pole — tento typ je používán pro pole statické velikosti. Pro jeho popis již neplatí Backus–Naurova forma zápisu. Délka se může opakovat a vytvoří se tak vícerozměrné pole.

```
typedef typ nový_typ[délka];
```

Výpis 5.13: Syntaxe pole

Sekvenční typ a pole musí nejdříve použít konstrukci `typedef`, aby se vytvořil nový typ. Ten se může dále používat standardním způsobem.

5.2 CORBA API

V této části bude vysvětlena pouze určitá množina programového rozhraní tak, aby bylo možné bez větších problémů pochopit výklad o testovací aplikaci. Některé části, jako je dynamické sestavování požadavku za běhu nebo přístup k jmenné službě, budou uvedeny až v konkrétní situaci na testovací aplikaci.

5.2.1 SII a SSI

Po sestavení definice v jazyce IDL popisující rozhraní budoucího objektu nastává čas použít IDL kompilátor. Ten vygeneruje dvě části aplikace, jež budou zajišťovat nízkoúrovňovou obsluhu objektu, který bude distribuován. Jedná se o `stub` a `skeleton`. Zde bych chtěl podotknout skutečnost, že `stub` v terminologii CORBA nemá stejný význam jako u již zmiňované technologie COM. Rozdíl je v tom, která část aplikace `stub` využívá. U technologie CORBA je použit na straně klienta, jenž využívá objekty známé v době překladu, zatímco u mechanizmu COM je `stub` potřebný na straně serveru, při použití komponent běžících v odděleném adresovém prostoru.

`Stub` a `skeleton` mohou být ve výsledné podobě oddělené[5] nebo naopak být obsaženy v jednom[6] souboru. Výsledný efekt je ve funkčnosti naprosto stejný. Na těchto generovaných částech je vidět otevřenost celého návrhu specifikace CORBA. Jelikož

[5] využito u implementace omniORB
[6] využito u implementace MICO

není pevně stanovena implementace, je zde možnost pro velký počet variací `stubu` a `skeletonu` při neměnném chování, ale různě výkonném kódu.

Každé rozhraní reprezentované třídou musí obsahovat jisté metody a třídy. Jejich implementaci vytvoří generátor sám. Jejich úkolem je umožnění základní manipulace s objektem. Pro následující popis vytvořím jednoduché rozhraní zobrazené ve výpisu 5.14 a na jeho základě se budu snažit vysvětlit funkci vygenerovaných částí.

```
interface A
{
   metoda1 (in long a);
   metoda2 (inout short b);
   metoda3 (out octet c);
};
```

Výpis 5.14: Vzorový příklad rozhraní

Jedny z prvků závazně obsažených v rozhraní jsou třídy reprezentující objektové reference. Klient využívající objekt nemůže uložit referenci do proměnné vytvořené standardním způsobem programování, v našem případě `A*`. Proto byly zavedeny ony třídy. V zásadě je dvojí přístup v použití referencí. První způsob nastává při použití třídy `A_var`. Reference přiřazená k `A_var` bude vlastněna právě instancí `A_var`. Ta je povinna referenci automaticky zrušit při svém zániku nebo při přiřazení jiné reference. Přístup tohoto druhu je velmi praktický pro svoji transparentnost při použití. Na druhou stranu existuje ještě druhý způsob využívající k uchování reference instanci typu `A_ptr`. Dosáhneme tak stejného chování jako při použití klasické proměnné typu `A*`. Některé implementace specifikují datový typ `A_ptr` pomocí konstrukce `typedef A* A_ptr`. Před destruováním instance `A_ptr` by měl programátor referenci uvolnit pomocí funkce `CORBA::Release(Object_ptr)`. Pro přístup k metodám přes instance typu `A_ptr` a `A_var` slouží operátor nepřímého přístupu.

Mezi objekty, které využívají koncept dědičnosti, musí být stanovena jistá pravidla, která dodávaná implementace musí sama zajistit. Obecně platí, že reference objektu potomka musí být automaticky zkonvertována na referenci jeho rodiče, ať již bezprostředního nebo vzdáleného. Toto pravidlo je uplatňováno ve vztahu k referencím uložených v proměnných typu `A_ptr` a `A_var`. Existují zde ovšem i jistá omezení. Pro konverzi zavedu označení **implicitní rozšíření** *(implicit widening)*.

Předpokládejme, že objekt `B` je odvozen od objektu `A`, pak implicitní rozšíření má definována následující pravidla:

1. reference na objekt potomka uložená v konstrukci `B_ptr` může být rozšířena na referenci uloženou v konstrukci `A_ptr`

2. reference na objekt potomka uložená v konstrukci `B_ptr` může být rozšířena na referenci uloženou v konstrukci `Object_ptr`
3. reference na objekt potomka uložená v konstrukci `B_var` může být rozšířena na referenci uloženou v konstrukci `A_ptr`
4. reference na objekt potomka uložená v konstrukci `B_var` může být rozšířena na referenci uloženou v konstrukci `Object_ptr`

Při porušení těchto pravidel implementace vyvolá běhovou chybu a aplikace musí být ukončena.

statické metody každého objektu

Z předcházejícího seznamu lze usoudit, že chybí možnost rozšíření referencí na typ `A_var`. Implicitní rozšíření pro tato pravidla skutečně neexistuje. Řešení tohoto problému se nabízí pomocí statické metody s předpisem `_duplicate(B_ptr)`.

V případě, že budeme potřebovat specializovat typ reference z předka na potomka, využijeme statickou metodu `_narrow(Object_ptr)`. Ta nám v případě neúspěchu vrátí referenci `nil` pomocí statické metody `_nil()`. Typicky se tato metoda využívá po prvním získání reference na objekt. Ta je vždy předána jako typ `Object`.

Pro ověření platnosti reference je určena funkce `CORBA::is_nil(Object_ptr)`. Ta vrací pravdivostní hodnotu `CORBA::Boolean`. Hodnota `false` je pro platnou referenci a hodnota `true` je pro referenci `nil`.

5.2.2 práce s řetězci

Pro práci s řetězci jsou definovány následující funkce:

1. `char* string_alloc( CORBA::ULong length );`
2. `char* string_dup( const char* );`
3. `void string_free( char* );`

První funkce je určena pro alokaci paměti pro řetězec určité délky. Druhá funkce je nejpoužívanější a má za úkol duplikaci celého řetězce. Poslední funkce uvolňuje alokovanou paměť.

5.2.3 reference

Pro získání reference na vzdálený objekt se většinou využívá IOR reference. Jedná se o řetězec, který jednoznačně identifikuje vzdálený objekt. Pro převedení IOR reference na objekt se využívá metoda ORBu

```
Object_ptr string_to_object(const char*);
```

Opakem této funkce je `object_to_string`. Po převedení reference je potřeba vždy specializovat typ statickou metodou `_narrow`.

Příkladem může být IOR reference, která produkuje naše komponenta UserAuth, jež je popsána dále.


```
IOR:000000000000001149444c3a55736572417574683a312e30000000
00000000010000000000000060000102000000000a3132372e302e302e
0310082f100000019afabcb000000000217949013000000008000000000
0000000a000000000000001000000010000002000000000000100010000
000205010001000100200001010900000000100010100
```


Tento řetězec obsahuje následující informace. Ty byly získány programem `iordump`.

```
Repo Id:  IDL:UserAuth:1.0

IIOP Profile
    Version:  1.2
    Address:  inet:127.0.0.1:33521
   Location:  corbaloc::1.2@127.0.0.1:33521/%af%ab%cb%00%00%00\
%00%02%17%94%90%13%00%00%00%08%00%00%00%00%00%00%00%00%0a
 Components:  Native Codesets:
              normal: ISO 8859-1:1987; Latin Alphabet No. 1
                wide: ISO/IEC 10646-1:1993; UTF-16, UCS\
 Transformation Format 16-bit form
              Other Codesets:
                      X/Open UTF-8; UCS Transformation Format\
 8 (UTF-8)
                      ISO 646:1991 IRV (International\
 Reference Version)
              Other Wide Codesets:
                      ISO/IEC 10646-1:1993; UCS-2, Level 1
        Key:  af ab cb 00 00 00 00 02 17 94 90 13 00 00 00 08
              00 00 00 00 00 00 00 00 0a
```

Kapitola 6

Testovací aplikace

Součástí této bakalářské práce je testovací aplikace. Ta by měla ukázat základní použití technologie CORBA. Aplikace se skládá ze dvou částí. První část je server, který bude poskytovat informace o aktuálním síťovém nastavení operačního systému Linux s jádrem řady 2.4 a druhá část aplikace je klient, který bude tyto informace přijímat. Minimální požadavky pro spuštění spolu s uživatelskou příručkou pro použití jsou uvedeny v příloze B na straně 62.

Obecně by se mohl vývoj jakékoli aplikace postavené na této technologii rozdělit do následujících etap:

1. zvolení vhodné implementace CORBA a konkrétního implementačního jazyka
2. zápis rozhraní objektů v jazyce IDL
3. vygenerování kostry aplikace kompilátorem jazyka IDL
4. návrh klientské aplikace v implementačním jazyce
5. implementace objektů a hlavního programu serveru

6.1 výběr implementace technologie CORBA

Výběr implementace závisí na mnoha okolnostech. Tento krok je stěžejní a jeho podcenění může způsobit větší či menší problémy při pozdějším vývoji softwaru. Obecně by se dal výběr implementace řídit několika kroky:

1. Implementace by měla být volena podle toho, který operační systém chceme použít. Zde by měla být použita implementace dodržující standard alespoň verze 2.1. Od této verze je již definován protokol IIOP, pomocí kterého mohou komunikovat přes protokoly z rodiny TCP/IP různé implementace provozované na rozličných platformách.

2. Výběr by měl být proveden na základě programovacího jazyka, ve kterém bude vyvíjený software programován. Bylo by dobré, aby byla k dispozici možnost rozšíření podporovaných programovacích jazyků. Eliminujeme tím pořizování další implementace v případě nutnosti vývoje pomocí jiného programovacího jazyka.

3. Neméně důležitou částí jsou objektové služby (COSS). Implementace by měla obsahovat alespoň ty z nich, jež jsou popsané ve standardu a měla by být rozšiřitelná o služby dodávané třetí stranou.

Před vlastním vývojem testovací aplikace jsem volil mezi implementací MICO[1] a implementací omniORB, které jsou šířeny pod licencí GPL. Obě implementace podporují operační systém Linux a jazyk C++, který bude použit při vývoji. Implementace MICO údajně vyhovuje standardu verze 2.3 a implementace omniORB vyhovuje standardu verze 2.4, čímž by se dalo usoudit, že implementace omniORB by byla vhodnější. Nicméně při psaní této práce byla implementace omniORB teprve k dispozici ve verzi *beta*, a proto jsem nakonec zvolil implementaci MICO, která je již stabilní. Vlastnosti obou implementací jsou shrnuty v tabulce 6.1.

6.1.1 ekonomická náročnost

Výběrem použitého softwaru pro vývoj testovací aplikace byl dosažen nulový stav nákladů. Veškerý software je šířen pod licencí GPL nebo jinými jí podobné. Musíme si ale uvědomit, že pro komerční použití volně dostupné implementace CORBy nebudou stačit. Právě nutnost koupě licence je jednou z nejdražších položek celého vývoje. Celá situace je komplikovaná ještě skutečností, že licence je někdy nutná pro každou vyvinutou komponentu. U komerčních produktů je možno získat celou škálu standardizovaných služeb COSS. Další položky, které zvyšují náklady, jsou platy softwarových analytiků a programátorů, náklady na hardware a operační systém.

[1] MICO je rekurzivní zkratka, která se rozvine v *MICO is CORBA*

Implementace	MICO	omniORB
Verze	2.3.7	4.0 beta 2
Specifikace	2.3	2.4
Operační systém	Linux MS Windows	Linux MS Windows
Jazyk	C++ Python	C++
COSS	Naming Event Externalization LifeCycle Relationship Time Trading	Naming

Tabulka 6.1: Přehled vlastností implementací MICO a omniORB

6.2 návrh aplikace

Pro návrh aplikace byl použit modelovací jazyk UML, viz [UML-01] a [UML-02]. Pro jeho grafické vyjádření byla vybrána aplikace DIA dostupná na `<http://www.lysator.liu.se/~alla/dia/>`. Celkový návrh je možno vidět v příloze D na straně 66. Z celkového návrhu je implementována pouze část, jež by měla být dostačující pro demonstrování možností architektury CORBA.

6.3 popis implementace aplikace

6.3.1 server

Serverová část je rozdělena na dvě komponenty. První z nich je ověřovací. Ta má za úkol ověřit identitu uživatele. Druhá komponenta má za cíl distribuovat jednotlivé informace o síťovém nastavení jádra hostitelského systému.

ověřovací komponenta

Ověřovací komponenta je napsána v programovacím jazyce Java. Je použita verze od firmy Sun Microsystems. V ní je obsažena i implementace technologie CORBA, která je v komponentě využívána. Implementace zde nebude popsána, jelikož se v principu jedná o stejné úkony jako při použití jazyka C++. Uvedu pouze jednoduché rozhraní, na které se budu dále odvolávat.

```
interface UserAuth
{
    boolean login(in string name, in string password);
};
```

Výpis 6.1: Rozhraní UserAuth.idl

Hlavním úkolem komponenty je ověření identity uživatele. Její jediná metoda `login` přijímá na vstupu dva parametry, které reprezentují jméno a heslo. V případě ověření totožnosti je vrácena hodnota `true`, jinak `false`.

informativní komponenta

Po vytvoření návrhu komponenty je zapotřebí jej přepsat pomocí jazyka IDL. Tím nám vznikne rozhraní nezávislé na implementačním jazyku. Textový soubor s rozhraním má většinou příponu `idl`. Dále následuje krok, který nám zajistí vygenerování kostry programu. Použijeme k tomu překladač jazyka IDL následujícím způsobem:

```
idl --no-poa --boa netinfo.idl
```

Tím jsme zajistili vytvoření stubu a skeletonu. Kostra programu bude upravena pro objektový adaptér BOA. Implementace MICO umisťuje kostru do jediného souboru. Vznikne soubor `netinfo.cc` a k němu odpovídající hlavičkový soubor `netinfo.h`. Hlavičku `netinfo.h` musíme vložit do našeho programu spolu s `CosNaming.h` hlavičkou, která je potřebná pro jmennou službu.

Nám stačí implementovat všechny objekty, jež byly deklarovány popisem rozhraní. Jedním způsobem jak toho docílit, je pomocí dědičnosti. Ukážeme si to na nejvrchnějším objektu `Base` našeho rozhraní.

Třídu, která bude implementovat naše rozhraní si můžeme nazvat dle libosti, důležité však je, aby byla odvozena od skeletonu rozhraní. V našem případě zvolíme `Base_implementace` pro název a odvodíme ji od skeletonu `Base_skel`, viz výpis 6.2.

```
class Base_implementace : virtual public Base_skel
{
public:
  Base_implementace(){}
  Net_ptr login(const char* name, const char* password);
};
```

Výpis 6.2: Implementace rozhraní Base

Tato třída nevyužívá žádný kód konstruktoru, tudíž jeho deklarace by zde nebyla ani nutná. Já ho zde ovšem uvádím z informativních důvodů. Pro implementaci všech operací jsou použity běžné konstrukce a operace jazyka C++, tudíž je nemá cenu zde uvádět. Za zmínku stojí implementace metody `login` zobrazená na výpisu 6.3.

Tato metoda využívá ověřovací komponentu k identifikaci uživatele. Metoda `login` je zde popisována, protože používá dynamické sestavování požadavků za běhu *(DII)*, což není běžné. Nejprve přiřadíme do proměnné `orb`, již inicializovanou instanci ORBu. Po té načteme IOR referenci identifikující naši ověřovací komponentu a převedeme ji metodou ORBu `string_to_object` na referenci na objekt. Převodem dojde k automatickému navázání spojení pomocí orbu s ověřovací komponentou.

Dále začíná na řádku 18 sestavování dynamického požadavku. Nejprve si vyžádáme od ověřovací komponenty metodu `login`. Pokud z nějakého důvodu, komponenta danou metodu nepodporuje, vrátí se hodnota `_nil`. V případě úspěšného předchozího volání můžeme začít se sestavováním požadavku. Každou referenci, než s ní budeme poprvé pracovat, bychom měli otestovat její platnost pomocí funkce `CORBA::is_nil`. Do požadavku přidáváme postupně argumenty, ty jsou vždy typu `any`. Argument musíme označit správným identifikátorem, který je očekáván na straně distribuovaného objektu. Uložení dat do typu `any` je provedeno pomocí kroků, které jsou popsány na straně 41. Pokud vzdálená metoda vrací nějaká data, musíme jejich typ specifikovat pomocí konstanty `TypeCode`, v našem případě `_tc_boolean`. Tím je celý požadavek sestaven a můžeme ho vyvolat za pomoci metody `invoke`. Návratová hodnota je opět uložena v typu `any`. Následně můžeme referenci na objekt uvolnit funkcí `CORBA::release`, protože jej již nebudeme používat. Jestliže návratová hodnota je rovna `true`, vytvoříme instanci objektu implementujícího rozhraní `Net` a referenci na ní pomocí statické metody `Net::_duplicate`, kterou následně budeme vracet jako výsledek metody `login`. V opačném případě, kdy autentifikace byla neúspěšná, vrátíme nulovou referenci `_nil`.

Jak je vidět, dynamické sestavovaní požadavků porušuje transparentnost distribuovaného programování. Většinou se využívají metody, které jsou umístěny ve stubu. Pak se vždy jedná o klasický styl volání metod. Jediné co musí být dodrženo, je alokační konvence pro parametry metod, viz tabulka 5.1 na straně 39.

```
1 Net_ptr Base_implementace::login(const char* name, const char*
    password)
2 {
3   CORBA::ORB_var orb = _orb();
4
5   ifstream in ("/tmp/UserAuth.ior");
6   char reference[1024];
7   in >> reference;
8   in.close();
9
10   CORBA::Object_ptr objekt = orb->string_to_object(reference);
11
12   if(CORBA::is_nil(objekt)==true)
13   {
14     cerr<<"Nemohu navazat spojeni s~overovaci komponentou"<<
         endl;
15      return Net::_nil();
16   }
17
18  CORBA::Request_ptr req = objekt->_request("login");
19   req->add_in_arg("name")<<=CORBA::Any::from_string(name,
       strlen(name));
20   req->add_in_arg("password")<<=CORBA::Any::from_string(
       password,strlen(password));
21   req->set_return_type(CORBA::_tc_boolean);
22   req->invoke();
23   CORBA::Boolean result;
24   req->return_value()>>=CORBA::Any::to_boolean(result);
25
26   CORBA::release(objekt);
27
28   if(result==true)
29      {
30   array[0]=Net::_duplicate(new Net_implementace);
31   return (array[0]);
32      }
33   return Net::_nil();
34 }
```

Výpis 6.3: Implementace metody login

Po implementaci všech rozhraní zbývá již jen inicializace celého systému. Kód k tomu potřebný je vyobrazen na výpisu 6.4 na straně 55.

Prvním krokem je inicializace ORBu. Ta je vykonána funkcí `ORB_init`. Jejími parametry jsou počet a seznam předaných parametrů při spuštění programu a identifikátor určující implementaci ORBu. Dále je inicializován objektový adaptér BOA pomocí metody `BOA_init` právě inicializovaného ORBu. Její parametry jsou opět počet a seznam parametrů a dále identifikátor označující implementaci adaptéru. Veškeré parametry, jež daná implementace rozpozná, jsou odebrány ze seznamu. Po inicializaci můžeme vytvořit instanci objektu reprezentující nejvýše postavené rozhraní celé komponenty. Instance se vytváří standardní cestou pomocí operátoru `new`.

Pro exportování reference na objekt použijeme místo IOR reference jmennou službu. Referenci na objekt reprezentující jmennou službu získáme po zavolání metody `resolve_initial_references` na instanci ORBu. Po specializaci typu statickou metodou `_narrow` můžeme zaznamenat v jmenné službě naši referenci na objekt implementující rozhraní `Base`. Záznam provede metoda `rebind`, která přebírá dva parametry. Prvním je struktura obsahující jméno a druh objektu. Druhým parametrem je pak reference na distribuovaný objekt.

Na řádku č. 17 je zaslán požadavek objektovému adaptéru, aby začal zpracovávat příchozí požadavky. Posledním krokem je vstoupení programu do smyčky událostí *(event loop)*. Z této smyčky již program může vystoupit pouze při volání metody `shutdown` na instanci orbu, který smyčku obsluhuje. Náš program takovou konstrukci neobsahuje, nicméně při jejím použití jsme povinni instanci distribuovaného objektu uvolnit pomocí funkce `release`.

6.3.2 klient

Pro tvorbu klienta můžeme použít dvě varianty. Klient může být určen pro textovou konzoly anebo grafické rozhraní. Druhá možnost je vhodnější pro uživatele a rozhodně přijatelnější i pro programátora. Náš klient bude rovněž grafický. Jeho zobrazovací funkce jsou ošetřeny pomocí knihovny QT, dostupné na `<http://www.troll.no>`.

Obecně platí, že GUI rámec *(framework)*, mezi který můžeme zařadit i knihovnu QT, je možné propojit s technologií CORBA třemi způsoby. Při použití GUI rámce se program uvede do nekonečné smyčky událostí, během níž čeká na vnější událost. Zabrání se tak efektu *polling*, kde jsou neustále využívány zdroje systému, aniž by to bylo třeba. Podobně se také chová program napsaný v technologii CORBA. Ten také vstoupí do smyčky událostí. Nicméně to je pravdivé pokud bychom psali grafický server. My však použijeme kombinaci událostních smyček, aby bylo vidět, jak snadno lze provést spojení QT s CORBou, aniž by bylo nutné specifikovat, zda náš klient bude poskytovat sám nějaké služby.

```
1 int main(int argc, char **argv)
2 {
3   CORBA::ORB_var orb = CORBA::ORB_init(argc, argv, "mico-local-
      orb");
4   CORBA::BOA_var boa = orb->BOA_init(argc, argv, "mico-local-
      boa");
5
6   Base_implementace* server = new Base_implementace;
7
8    CORBA::Object_var nsobj = orb->resolve_initial_references("
       NameService");
9
10    CosNaming::NamingContext_var nc = CosNaming::NamingContext::
        _narrow(nsobj);
11    CosNaming::Name name;
12    name.length(1);
13    name[0].id = CORBA::string_dup("Base");
14    name[0].kind = CORBA::string_dup("");
15    nc->rebind(name, server);
16
17    boa->impl_is_ready(CORBA::ImplementationDef::_nil());
18
19    orb->run();
20    CORBA::release(server);
21
22   return 0;
23 }
```

Výpis 6.4: Inicializace systému

1. Dispatcher — slouží pro kombinaci událostních smyček. Tento způsob bude používat naše aplikace.
2. Multithreadový ORB — pomocí vláken lze zabránit kolizi u smyček událostí. Lze využít například implementaci omniORB.
3. standardní způsob spojení — aplikace, která nebude poskytovat svému okolí žádné služby, může využít standardní způsob spojení známý z textové konzole.

Nejdůležitější částí z celé klientské části aplikace je navázání spojení. Ta je vyobrazena na straně 57. Po získání reference na objekt a po specializaci typu se již volají metody klasicky pomocí operátoru nepřímého přístupu. Kromě standardních hlaviček potřebných pro knihovnu QT, je nutné vložit hlavičky `qtmico.h` *(dispatcher)*, `CosNaming.h` *(jmenná služba)*, `netinfo.h` *(stub naší komponenty)*.

Nejprve je nutné vytvořit instanci třídy `CApp`. Ta bude sloužit jako dispatcher. Její vytvoření automaticky inicializuje ORB a adaptér BOA na základě předaných vstupních parametrů programu. Následuje získání reference na jmennou službu a sestavení požadavku, který bude charakterizovat hledaný objekt. Tato část je stejná jako na straně serveru. Na řádku č. 18 je vyslán požadavek funkcí `resolve` na nalezení objektu `Base` pomocí jmenné služby. Nalezená reference na objekt bude vrácena do proměnné `object`. Dále musíme provést specializaci typu statickou metodou `_narrow`. Vrácenou referenci můžeme následně distribuovat skrze naší aplikaci. Jelikož je ale uložena v proměnné typu končícím sufixem `_var`, musíme ji vždy duplikovat statickou metodou `_duplicate`. Tím zabráníme porušení vlastnictví reference. Na konci funkce `main` vstoupíme do smyčky událostí (`return appl.exec()`), kterou bude obsluhovat MICO dispatcher.

```
1 #include <mico/qtmico.h>
2 #include <mico/CosNaming.h>
3 #include "netinfo.h"
4
5 int main(int argc, char **argv)
6 {
7   CApp appl(argc, argv);
8
9    CORBA::Object_var nsobj = appl.orb->
        resolve_initial_references("NameService");
10
11    CosNaming::NamingContext_var nc = CosNaming::NamingContext::
        _narrow(nsobj);
12    CosNaming::Name name;
13
14    name.length(1);
15    name[0].id = CORBA::string_dup("Base");
16    name[0].kind = CORBA::string_dup("");
17
18    CORBA::Object_var object = nc->resolve(name);
19
20    Base_var server_base = Base::_narrow(object);
21
22   fMain* mw= new fMain(Base::_duplicate(server_base));
23   appl.setMainWidget(mw);
24   mw->show();
25   return appl.exec();
26 }
```

Výpis 6.5: Inicializace spojení na straně klienta

Kapitola 7

Závěr

Tato práce se zabývala problematikou distribuovaných objektových systémů se zaměřením na technologii CORBA. Základním přínosem se stal ucelený pohled na celou problematiku. Práce by měla částečně usnadnit výběr vhodné technologie při vývoji distribuovaného softwaru. Současně poskytuje popis logické stavby technologie CORBA. Větší měrou zde byla shrnuta problematika C++ mapování pro IDL jazyk.

Dalším cílem, kterého jsem se snažil dosáhnout, byl návrh jednoduché aplikace založené na architektuře CORBA. Záměrem bylo dokázat jak hardwarovou, tak i softwarovou nezávislost této technologie. Proto byly při vývoji aplikace použity dvě implementace CORBy a dva různé programovací jazyky. Celá aplikace prošla všemi základními stupni vývoje, které se uskutečňují v běžné praxi. Tato aplikace spolu s touto prací vytváří jakousi základní pomůcku pro budoucí studium architektury CORBA.

Problematika distribuovaných systémů je velmi složitá a je využívána větší či menší měrou při vývoji aplikací. Z programátorského hlediska je zajímavá a skoro každý běžný uživatel s distribuovanými systémy pracuje při každodenní činnosti, aniž by o tom věděl.

Do budoucna se dají očekávat velké změny, které jsou již částečně uskutečňovány na základě specifikace CORBA 3. Mělo by dojít k velkému sblížení s architekturou EJB. Nic ale na tom nemění, že technologie CORBA je i bude jednou z nejvýznamnějších v heterogenních prostředích.

Literatura

[DOB-01] ZELENÝ, J., NOŽIČKA, J. *COM+, CORBA, EJB.* 1. vyd. Praha : BEN, 2002. 312 s. ISBN 80-7300-057-1.

[DOB-02] PUDER, A., RÖMER, K. *MICO — An Open Source CORBA Implementation.* 3. vyd. San Francisco : Morgan Kaufmann Publishers, 2000. 195 s. ISBN 1-55860-666-1.

[DOB-03] ORFALI, R., HARKEY, D., EDWARDS, J. *Instant CORBA.* 1. vyd. New York : Wiley Computer Publishing, 1997. 313 s. ISBN 0-471-18333-4.

[DOB-04] SPELL, B. *Java — Programujeme profesionálně.* 1. vyd. Praha : Computer Press, 2002. 1022 s. ISBN 80-7226-667-5.

[DOB-05] MATTHEW, N., STONES, R. *Linux — Programujeme profesionálně.* 1. vyd. Praha : Computer Press, 2001. 1079 s. ISBN 80-7226-532-6.

[DOB-06] CANTÙ, M. *Mastering Delphi 6.* 1. vyd. Alameda : Sybex Inc., 2001. 1071 s. ISBN 0-7821-2874-2.

[DOB-07] FARLEY, J. *Java Distributed Computing.* 1. vyd. Sebastopol : O'Reilly, 1998. ISBN 1-56592-206-9E.

[DOB-08] SWEET, D. *KDE 2.0 Development.* 1. vyd. Indianapolis : Sams Publishing, 2001. 540 s. ISBN 0-672-31891-1.

[DOB-09] ROSENBERGER, J. L. *Teach Yourself Corba in 14 Days.* b.m. : Sams Publishing, 1998. 500 s. ISBN 0672312085.

[DOW-01] Object Management Group. *CORBA Success Stories* [online]. Object Management Group. Dostupné na WWW: `<http://www.corba.org/success.htm>` [cit. 2003].

[DOW-02] MUKNŠNÁBL, J. *Gnome a SUN* [online]. Dostupné na WWW: `<http://www.reboot.cz/index.phtml?id=98>` [cit. 2003].

[DOW-03] GNOME. *Introduction to Bonobo* [online]. GNOME. Dostupné na WWW: `<http://www.gnome.org/gnome-office/bonobo.shtml>` [cit. 2003].

[DOW-04] KOPECKÝ, J. *SOAP — konečně správné pH pro váš software?* [online]. Dostupné na WWW: `<http://www.zive.cz/h/Programovani/Ar.asp?ARI=100819&CAI=>` [cit. 2002].

[DOW-05] LAMPA, P. *CORBA a IIOP* [online]. Dostupné na WWW: `<http://www.fit.vutbr.cz/~lampa/papers/corba.html>` [cit. 2003].

[DOW-06] SAI-LAI, L., RIDDOCH, D., GRISBY, D. *The omniORB User's Guide version 3.0* [online]. Dostupné na WWW: <http://omniorb.sourceforge.net/omni30/omniORB.pdf> [cit. 2002].

[OMG-01] Object Management Group, Inc. *The Common Object Request Broker Architecture and Specification* [online]. Object Management Group, Inc. Dostupné na WWW: <http://www.omg.org/cgi-bin/doc?formal/99-10-07.pdf> [cit. 2002].

[OMG-02] Object Management Group, Inc. *C++ Language Mapping Specification* [online]. Object Management Group, Inc. Dostupné na WWW: <http://www.omg.org/cgi-bin/doc?formal/99-07-41.pdf> [cit. 2002].

[OMG-03] Object Management Group, Inc. *CORBAservices: Common Object Services Specification* [online]. Object Management Group, Inc. Dostupné na WWW: <http://www.omg.org/cgi-bin/doc?formal/98-12-09.pdf> [cit. 2002].

[UML-01] SCHMULLER, J. *Myslíme v jazyku UML.* 1. vyd. Praha : Grada, 2001. 360 s. ISBN 80-247-0029-8.

[UML-02] PAGE-JONES, M. *Základy objektově orientovaného návrhu v UML.* 1. vyd. Praha : Grada, 2001. 368 s. ISBN 80-247-0210-X.

Příloha A

Přehled úspěšného použití

V této příloze bych rád uvedl úspěšné nasazení technologie CORBA v praxi. Seznam je převzat z internetové stránky organizace OMG[DOW-01]. Z následující tabulky lze vyvodit závěr, že veškeré aplikace jsou komplexní a jsou používány velkými korporacemi.

- **Ballistic Missile Defense Organization** — řízení přístupu k databázi tajných dat pro jednotlivé pověřené pracovníky
- **NASA Goddard Space Flight Center** — plánovací systém s podporou vesmírných zařízení
- **Continental Power Exchange** — telekomunikační systém pro obchod s elektrickou energií
- **The Weather Channel** — systém zpracovávající a poskytující satelitní informace o počasí
- **American Airlines** — reservační systém s rozloženou zátěží dle aktuální potřeby
- **Zuercher Kantonalbank** — systém určený pro oblast bankovnictví
- **Lufthansa Systems** — řízení leteckého provozu
- **Cisco Systems, Inc.** — obchodní systém

Příloha B

Uživatelská příručka

B.1 Požadavky

Na minimální provoz musí být k dispozici prostředky, které lze rozdělit do dvou skupin:

1. prostředky potřebné pro server
2. prostředky nutné pro klienta

Minimální požadavky shrnuje tabulka B.1.

	Server	**Klient**
Procesor	Pentium nebo ekvivalentní	Pentium nebo ekvivalentní
RAM	128 MB	128 MB
OS	Linux	Linux
Verze jádra	2.4.X	2.2.X
Překladač	GNU gcc 3.2 Sun JDK 1.4.1	GNU gcc 3.2
CORBA	MICO 2.3.7	MICO 2.3.7
Ostatní	Síťová karta (podpora TCP/IP)	Síťová karta (Podpora TCP/IP) Trolltech QT 3.0.6 MICO 2.3.7 QT dispatcher

Tabulka B.1: Minimální požadavky

Aplikace byla úspěšně testována na dvou počítačových sestavách, jak v lokálním, tak i síťovém prostředí. Parametry obou sestav jsou shrnuty v tabulce B.2 na straně 63.

Sestava	1	2
Procesor	Intel Pentium IV Northwood 1.8 GHz	Intel Pentium III Mobile 700Mhz
RAM	512 MB	128 MB
Operační systém	RedHat Linux 7.3	RedHat Linux 7.3
Verze jádra	2.4.20 *(Vanilla)*	2.4.18-3 *(RedHat)*
Překladač	GNU gcc 3.2 Sun JDK 1.4.1	GNU gcc 3.2
CORBA	MICO 2.3.7	MICO 2.3.7
QT	3.06	3.06

Tabulka B.2: Parametry použitých počítačových sestav

B.2 Překlad

Nejprve je nutné přeložit veškerý potřebný software. Překlad MICO, QT a GCC trvá na počítačové sestavě č. 1 přibližně 4 hodiny. Instalace se provádí na základě dokumentace příslušného softwaru.

Pro samotný překlad aplikací je použit nástroj `GNU make`. Jednotlivé aplikace jsou umístěny na CD v adresářích `server`, `userauth` a `client`. V každém adresáři se použije příkaz `make`, který bude řídit překlad. Předpokládá se, že MICO se bude nacházet v adresáři `/opt/mico`.

B.3 Spuštění

Pro spuštění lze následně využít v každém adresáři napsaných skriptů `start`. Skript zaručí správné spuštění aplikace pokud je volný port 12456. Nejprve je nutno spustit ověřovací komponentu `userauth`. Ta vygeneruje soubor `/tmp/UserAuth.ior`. Ten musí být distribuován jakýmkoli způsobem do adresáře `/tmp` na stanici, kde poběží server. Po té se již může spustit server. Zároveň s ním se automaticky spustí daemon jmenné služby a nakonec je možné spustit jakýkoli počet klientů.

Po spuštění klienta se v menu `tools/login` zadá jméno a heslo uživatele. V současné době je k dispozici jméno `root` a slovníkové heslo `linuxa`. Ty se předají na server, ten u ověřovací komponenty provede autentifikaci a v případě úspěchu se v klientu zobrazí okno s informacemi o jádře.

Pokud uživatel změní některou z položek a použije tlačítko `apply`, server tuto hodnotu v jádře systému nezmění, ale vypíše pouze informaci na chybový výstup. Aplikace se chová dále stejně, jako kdyby byly skutečně změněny parametry jádra.

Příloha C

CD–ROM

Na přiloženém kompaktním disku jsou umístěny:

- kompletní zdrojové kódy testovací aplikace
  /application/source
- binární forma testovací aplikace
  /application/binary
- zdrojové texty této bakalářské práce pro typografický systém LaTeX
  /text/source
- text této bakalářské práce ve formátech PS a PDF
  /text/final
- UML objektový návrh testovací aplikace v programu Dia a ve formátu EPS
  /application/uml
- použitý volně dostupný software pro nekomerční použití
  /tools

Příloha D

Objektový návrh