České Vysoké Učení Technické v Praze

Fakulta elektrotechnická

Katedra řídicí techniky

DIPLOMOVÁ PRÁCE

Prostředí pro vzdálenou vizualizaci

2003 Michal Vrubel

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, software atd.) uvedené v přiloženém seznamu.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu § 60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne: ....   Podpis: ....

Anotace

Diplomová práce popisuje návrh aplikačního prostředí pro vzdálenou vizualizaci. Prostředí je použitelné zejména pro vizualizaci v oboru průmyslové automatizace. Navrhované řešení využívá www prohlížeče a možnosti Java appletu jako rozhraní pro vizualizaci a umožňuje sledovat vzdálené technologické procesy. Prostředí pro vizualizaci využívá pro přenos technologických dat počítačové sítě. Jako technologie pro vlastní přenos dat je použita implementace architektury CORBA. Jako zdroj technologických dat se předpokládá použití OPC serveru. Teoretická část se zabývá popisem architektury CORBA. Přináší úvod pro její pochopení a také nezbytné poznatky potřebné pro implementaci CORBA aplikace. Praktická část popisuje vlastní návrh softwaru. Návrh je rozdělen do tří částí. První část popisuje návrh CORBA-OPC můstku, který představuje základní krok k propojení technologií CORBA a OPC. Druhá část popisuje návrh jednoduchého návrhového prostředí pro podporu návrhu vizualizace. Třetí část popisuje Java applet, realizující rozhraní pro vizualizaci. Funkce softwaru je demonstrována na úloze vizualizace třídicího mechanismu barevných míčků, který byl instalován v laboratoři katedry řídicí techniky, elektrotechnické fakulty ČVUT v Praze.

Annotation

The diploma thesis describes a design of a software for remote visualization. Software has been designed for purposes of visualization in branch of industrial automation. Here proposed solution presumes using of www browser as a platform for visualization interface and takes advantage of Java applets. Interface allows monitoring of remote technological processes. Solution uses CORBA architecture and CORBA network protocol for technological data transfer across computer network. As source for technological data is used OPC server. Teoretical part of diploma thesis describes CORBA architecture and brings introduction in CORBA and practical findings for developers of CORBA aplications. Practical part of diploma thesis describes implementation of that kind of software. Design is split in three parts. First part describes a design of CORBA-OPC bridge, which will be used for interconnection between CORBA and OPC. Second part describes a design and an implementation of a simple tool for creating visualization for technological processes. Third part describes an implementation of vizualization applet. Software is demonstrated in the visualization task for color ball sorter installed in laboratory of department of control engineering, faculty of electrical engineering, CTU in Prague.

Obsah

1 Úvod
2 Teoretická část - Architektura CORBA
2.1 Základní charakterisitka
2.2 Komunikační model
2.2.1 Objektová reference
2.3 Komponentový model
2.3.1 ORB
2.3.1.1 Metody pro inicializaci
2.3.1.2 Metody informující o dostupných službách
2.3.1.3 Metody pro komunikaci
2.3.1.4 Metody pro konverzi objektové reference
2.3.1.5 Metody pro kontrolu vláken a programového běhu orba
2.3.2 Objektový adaptér
2.3.2.1 Objektový adaptér POA (Portable Object Adapter)
2.3.2.2 Hierarchie POA adaptérů, vytváření POA
2.3.2.3 POA politiky
2.3.2.4 Vyhledání Servanta
2.3.2.4.1 Mapa aktivních objektů (AOM)
2.3.2.4.2 Politika Servant Retention Policy
2.3.2.4.3 Politika Request processing Policy
2.3.2.5 Aktivace Servanta
2.3.2.5.1 Explicitní aktivace
2.3.2.5.2 Aktivace podle potřeby
2.3.2.5.3 Implicitní aktivace
2.3.2.5.3.1 Politika Implicit Activation Policy
2.3.2.6 Deaktivace Servanta
2.3.2.7 Životnost Servanta
2.3.2.7.1 Politika LifespanPolicy
2.3.2.8 Politika Id Assignment Policy
2.3.2.9 Politika Object Id Uniqueness Policy
2.3.2.10 Management vláken v POA
2.3.2.10.1 Politika ThreadPolicy
2.3.2.11 POA manager
2.3.2.12 Aktivace POA adaptéru
2.3.3 Jmenná služba (NS)
2.3.3.1 Jmenný prostor v NS
2.3.3.2 Rozhraní jmenné služby
2.3.3.3 Jména, jmenné vazby, jmenné kontexty
2.3.3.4 Řetězcová jména
2.3.3.5 Služby NS
2.3.3.5.1 Vytvoření jmenného kontextu
2.3.3.5.2 Rušení jmenného kontextu
2.3.3.5.3 Listování jmenným kontextem
2.3.3.5.4 Vázání objektů
2.3.3.5.5 Odvázání objektů
2.3.3.5.6 Získávání objektů
2.3.3.5.7 Rozhraní BindingIterator
2.3.4 Statická komunikace klient-server

2.3.4.1 IDL Stub
2.3.4.2 IDL Skeleton
2.3.5 Dynamická komunikace klient-server
2.3.5.1 Rozhraní Dynamic Invocation Interface (DII)
2.3.5.1.1 Sestavení požadavku
2.3.5.1.2 Odeslání požadavku
2.3.5.1.3 Čekání na odpověď a příjem odpovědi
2.3.5.1.4 Destrukce požadavku
2.3.5.2 Dynamický skeleton - rozhraní Dynamic Skeleton Interface (DSI)
2.3.5.2.1 Předání požadavku objektu - rutina DIR
2.3.5.2.2 Zpracování požadavku klienta
2.3.5.3 Služba Interface Repository (IR)
2.3.5.3.1 Struktura IR
2.3.5.3.1 Vyhledávání a čtení v IR
2.3.5.3.3 Identifikace typů a rozhraní pomocí IR
2.3.5.3.4 Vytváření deklarací a zápis do IR
2.4 Jazyk OMG IDL
2.4.1 Úvod do IDL
2.4.2 Přehled klíčových slov jazyka IDL, pravidla zápisu identifikátorů
2.4.3 Komentáře
2.4.4 Konstanty
2.4.5 Základní datové typy, typ string a wstring
2.4.6 Strukturované typy, speciální typy
2.4.6.1 Záznam (struct)
2.4.6.2 Pole
2.4.6.3 Sekvence (sequence)
2.4.6.4 Typ Any (Any)
2.4.7 Obory platnosti identifikátorů
2.4.7.1 Jmenné prostory (module)
2.4.7.2 Rozhraní (interface)
2.4.8 Deklarace nového typu
2.4.9 Deklarace operace - metody
2.4.10 Parametry funkcí, návratový typ funkce
2.5 CORBA - Používaná hesla

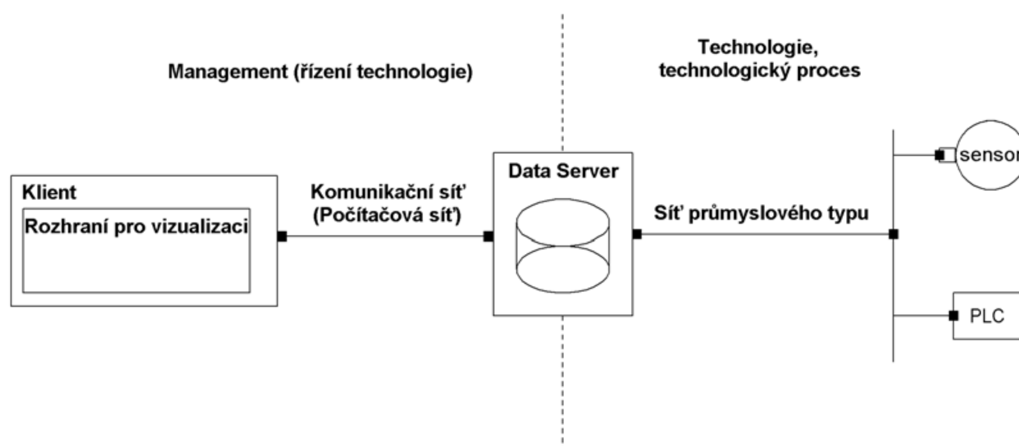
3 Praktická část - Implementace PPVV
3.1 CORBA-OPC můstek
3.1.1 Základní charakteristika
3.1.2 Použité technologie - shrnutí
3.1.2.1 COM a OPC
3.1.3 Podrobnosti implementace
3.1.3.1 CORBA a COM spolupráce
3.1.3.2 Serverové rozhraní můstku - IDL rozhraní
3.1.3.3 Práce s pamětí pro přenášená data
3.1.3.4 Vláknový model CORBA-OPC můstku
3.1.3.4.1 Komunikace vláken servanta a OPC klienta
3.1.3.4.2 Alokace vláken a 'timeout' pro klienty
3.1.4 Instalace a spouštění
3.1.5 Popis konzoly, ovládání aplikace
3.1.6 Testování CORBA-OPC můstku

3.1.7 Implementace CORBA serveru a klienta v prostředí Borland C++ Builder 5.0
3.2 Návrhové prostředí pro vizualizaci
3.2.1 Základní charakteristika, použité technologie
3.2.2 Instalace a spouštění
3.2.3 Popis návrhového prostředí
3.2.3.1 Práce v návrhovém prostředí
3.2.3.1.1 Vytvoření návrhu
3.2.3.1.2 Práce s komponentami
3.2.3.1.3 Uložení návrhu
3.2.3.1.4 Export pro www
3.2.3.2 Vytváření komponenty
3.2.3.2.1 Implementace komponenty
3.2.3.2.2 Třída ComponentFactory
3.2.3.2.3 Příklad
3.2.3.3 Další možnosti vývoje
3.2.4 Překlad zdrojového kódu
3.3 Vizualizační applet
3.3.1 Základní charakteristika, použité technologie
3.3.2 Úvod do problematiky appletu
3.3.2.1 Charakteristika appletu
3.3.2.2 Životní cyklus appletu
3.3.2.3 Konstruktor appletu
3.3.2.4 Vykreslování appletu
3.3.2.5 Nahrávání a spouštění appletu v prohlížeči
3.3.2.6 Archívy JAR
3.3.2.7 Předání parametrů appletu
3.3.2.8 Bezpečnostní omezení
3.3.3 Podrobnosti Implementace
3.3.4 Popis prostředí appletu
3.3.5 Ovládání appletu
3.4 Vizualizace třídícího mechanismu barevných míčků
3.5 Obsah CD-ROM, organizace zdrojového kódu
3.5.1 Diplomová práce v elektronické podobě (doc, pdf)
3.5.2 Binární (spustitelný) kód aplikací
3.5.3 Zdrojový kód aplikací

4 Závěr
5 Reference

1 Úvod

V rámci diplomové práce jsem měl za úkol vytvořit jednoduché aplikační prostředí pro vzdálenou vizualizaci (PPVV). Aplikace pro vzdálenou vizualizaci mají obvyklé použití v podnikové sféře v průmyslové automatizaci, kde se uplatňují pro přenos technologických dat ze zdroje (technologie, technologický proces) a jejich vizuální interpretaci, nejčastěji pro účely dohledu nad průběhem výroby, kontroly kvality a podobně. Prostředí pro vzdálenou vizualizaci pracuje v prostředí s distribuovanou architekturou. Zde navrhované řešení je určeno pro podporu výuky na katedře řídicí techniky. Na katedře již nějakou dobu probíhají snahy zpřístupnit část výpočetních prostředků pro studijní účely studentům na síti a využít možnosti sítě internet pro vzdálenou výuku. Ve své práci jsem měl jako distribuovaného prostředí využít konvenční počítačové sítě (internet). Základní hardwarovou a softwarovou koncepci PPVV vidíme na obr. 1. Koncepci tvoří server a klient v počítačové síti, kde klient je vybaven rozhraním pro vizualizaci technologických dat. Úkolem serveru je sběr dat z technologie a jejich zpřístupnění klientům.



Obr. 1: Základní koncepce prostředí pro vzdálenou vizualizaci

Tato koncepce je však pro vlastní návrh PPVV poněkud obecná. Před vlastním návrhem bylo potřeba se zabývat výběrem technologie použité při návrhu. Z požadavku zpřístupnění rozhraní pro vizualizaci na síti internet se v podstatě přímo nabízelo využít možnosti www prohlížeče. Jako základní běhové prostředí pro rozhraní pro vzdálenou vizualizaci bylo tedy použito www prohlížeče, pro který byl navržen odpovídající software. Pro dané využití, www prohlížeč nabízí řadu užitečných vlastností. Současné www prohlížeče, kromě možnosti zobrazit holý hypertextový dokument, obvykle umožňují také zobrazit počítačovou grafiku a různé multimediální prvky. Kromě toho, umožňují také interpretovat různorodé počítačové jazyky, které můžeme využít pro návrh vlastního software. Z těchto jazyků je také dostupná pro využití řada moderních výpočetních technologií. Jeho nespornou předností je také skutečnost, že ho není obvykle potřeba zvlášť instalovat. Dnes bývá www prohlížeč obvyklou součástí běžně používaných operačních systémů. Stejně tak obvykle není potřeba instalovat aplikace spouštěné v prohlížeči jako jsou applety nebo ActiveX komponenty. Obvyklý postup spočívá v on-line stažení na hostitelský počítač a následném spuštění. Pro uživatele může být tato vlastnost užitečná (pokud např. nedisponuje právy správce počítače).

Pro implementaci rozhraní pro vizualizaci byla použita technologie Java appletů. Tuto možnost jsem zvolil podle míry poskytnuté technologické základny a s

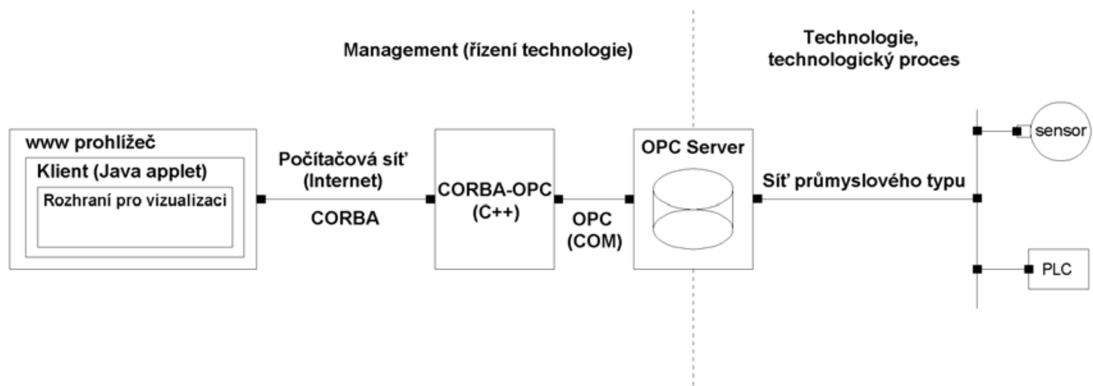
přesvědčením, že tato technologie je pro daný účel adekvátní. V současnosti se technologie Java appletů pro internetová řešení široce využívá a těší se značné obliby. Java je pro daný účel vhodný jazyk, nabízející koncepty potřebné pro implementaci PPVV. Kromě jiného, podporuje síťové programování, práci s grafikou a návrh grafického uživatelského rozhraní. Java také umožnila dosáhnout nezávislosti na platformě a umožňuje vizualizaci spouštět jak ve Windows, tak např. v Linuxu.

Prostředí pro vizualizaci navržené v rámci diplomové práce využívá jako zdroje technologických dat OPC server. Aplikační rozhraní OPC (*OLE for Process Control*) bylo navrženo a standardizováno, viz. [12], pro jednotný přístup aplikací ke zdroji technologických dat a bylo optimalizováno pro potřeby průmyslové automatizace. Zdroji technologických dat jsou obvykle zařízení používaná ve výrobě, zpravidla se jedná o různá čidla, senzory, jednotky řízení, PLC a podobně. V minulosti se často řešil problém s výběrem vhodné přenosové technologie pro sběr dat z čidel. Jelikož neexistoval jednotný standard, obvykle aplikace typu PPVV neposkytovaly podporu pro širokou škálu koncových zařízení. Tento problém byl vyřešen až s příchodem protokolu DDE a OPC. DDE je oproti OPC poněkud dřívější technologie, ačkoli dnes má poměrně široké použití. Má však určité nedostatky, původně totiž nebyla určena přímo pro průmyslovou automatizaci. V mé práci jsem se rozhodl využít OPC, které má dnes také poměrně širokou podporu a navíc se jedná o otevřený standard, optimalizovaný pro úlohy průmyslové automatizace. Podporu pro OPC nabízí také většina PLC používaných v laboratoři katedry (včetně PLC WAGO používaného v úloze třídění barevných míčků, na které byly provedeny závěrečné testy softwaru). Předpokládám, že současný trend používání OPC bude i nadále pokračovat a je možné že v budoucnosti bude pro většinu průmyslových zařízení zpřístupňující data existovat odpovídající OPC server. Díky tomu bude možné v PPVV zpřístupnit data těchto zařízení.

Aplikace PPVV je aplikace typu klient-server. Jelikož klient byl implementován v jazyce Java, bylo potřeba také řešit problém absence možnosti přistupovat v Javě k technologii COM. Tuto technologii využívá OPC. Java bohužel neobsahuje standardní API pro její podporu. Java však umožňuje využít COM nepřímo prostřednctvím technologie CORBA. CORBA umožňuje propojit dvě aplikace nezávisle na implementačním jazyce a platformě. Popis distribuované architektury CORBA, vzhledem k jejímu významu, byl začleněn do teoretické části diplomové práce v kapitole 2. V současnosti existuje CORBA implementace jak pro jazyk Java, tak pro jazyk C++. V tomto jazyce byl jako součást PPVV navržen můstek mezi technologiemi OPC a CORBA. Jeho popis je uveden společně s ostatními aplikacemi PPVV v kapitole 3.

Součástí PPVV je také jednoduchý nástroj pro podporu návrhu vizualizace. Cílem práce v tomto nástroji je vytvořit grafické schéma vizualizovaného technologického procesu. Tento nástroj umožňuje kromě základních funkcí jednoduchý export vytvořeného návrhu spolu s appletem realizujícím rozhraní pro vizualizaci na libovolný www server. Studenti mohou nástroj rozšiřovat o nové komponenty a zlepšovat tak v PPVV možnosti vizualizace. Pro implementaci tohoto návrhového prostředí byl také využit jazyk Java. Popis této aplikace je uveden v odst. 3.2.

Na základě výběru technologií se koncepce PPVV strukturalizovala do podoby, kterou vidíme na obr. 2.



Obr. 2: Výchozí koncepce prostředí pro vzdálenou vizualizaci

2 Teoretická část - Architektura CORBA

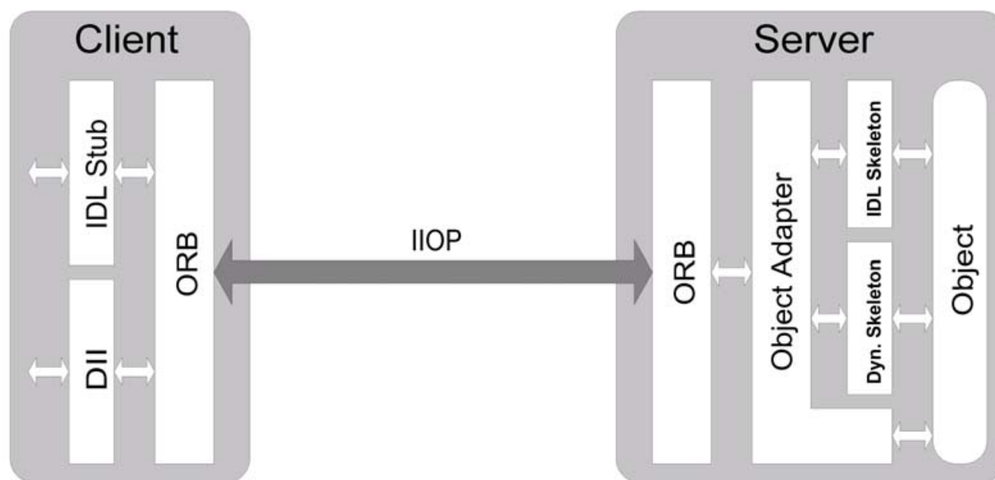
2.1 Základní charakteristika

Corba byla vyvinuta pro potřeby návrhu objektově orientovaného software s velkým důrazem kladeným na distribuované koncepty. Hlavním cílem bylo vytvořit jednotnou architekturu pro návrh objektově orientovaného software, prostřednictvím které by bylo možné dosáhnout potřebné součinnosti softwaru v síťovém prostředí a nezávislosti na implementačním jazyce a použité platformě. *Corba* nachází hlavní uplatnění při návrhu distribuovaných aplikací, tj. v prostředí, ve kterém jsou jednotlivé části aplikace roztroušeny v síti (rozděleny na samostatné výpočetní kontexty) a jež navzájem spolupracují a komunikují prostřednictvím datové výměny v síti. Významného uplatnění nalezne zejména při návrhu informačních systémů pracujících v prostředí internetu. Kromě běžných a základních konceptů jako je koncepce síťové komunikace v sobě *corba* integruje řadu doplňkových vlastností a služeb, jež vývojáři využijí při návrhu distribuovaného softwaru (zabezpečení, transakce, životní cykly objektů, jmenná služba a další). *Corba* se vyznačuje vysokým stupněm slučitelnosti se stávajícími technologiemi (programovací jazyky, různé operační systémy, ...) a maximálním stupněm slučitelnosti aplikací vytvořených v *corbě*. Vývojem a standardizací *corby* se zabývá organizace *OMG* (*Object Management Group*), viz. [11]. Obsahem jednotlivých částí kapitoly o *corbě* bude popis těch komponent, které jsem použil při návrhu, a které jsem pokládal za důležité z hlediska pochopení struktury a modelu komunikace v *corbě*. Seznam hesel používaných v textu, společně s jejich významem, je uveden v odst. *2.5 CORBA - Používaná hesla*.

2.2 Komunikační model

V *corbě* se uplatňuje převážně komunikace typu klient - server. V tomto typu komunikace si účastníci vyměňují informace na základě požadavků a odpovědí. Požadavky vysílají klienti na server a ten požadavky zpracovává a výsledek zpracování zasílá zpět klientům ve formě odpovědí. V objektové corbě to má svou specifickou podobu. Na serveru existuje jeden nebo více *objektů*, tzv. *objektových implementací*, které pomocí svých metod implementují služby. Klienti adresují požadavky *objektům* a server zpracovává žádost tak, že volá odpovídající metodu objektu. V distribuované aplikaci vytvořené v corbě se na komunikaci podílí několik komponent, které jsou součástí její architektury (viz. odst. 2.3 Komponentový model). Obrázek 3 znázorňuje schéma komunikace *klienta* a *serveru* včetně základních komponent corby, které se komunikace zúčastňují.

Klient může ke komunikaci použít buď tzv. *IDL Stub* anebo rozhraní *Dynamic Invocation Interface* (*DII*). *IDL Stub* a *DII* napomáhají sestavit požadavek a ten se pak předá *orbu*. *Orb* požadavky přeloží z objektové formy do formy *IIOP* zpráv a zajišťuje vlastní odeslání a příjem zpráv. *Klient* má možnost řídit pouze obecný princip komunikace. Může např. určit, zda bude chtít zasílat více zpráv najednou a nebo všechny zprávy sekvenčně, zda požaduje odpověď a zda bude na odpověď aktivně čekat anebo zda odešle požadavek s tím, že bude pokračovat ve své činnosti a *klient* se bude dotazovat na odpověď v průběhu své činnosti atp. *Server* obdobně používá buď tzv. *IDL Skeleton* anebo tzv. *dynamický skeleton*. Metody *skeletonu* volá *orb*, který skeletonu předává zprávy od klienta.



Obr. 3: Schéma komunikace klient - server v corbě

Objektový adaptér pomůže *orbu* vyhledat objekt, kterému je zpráva adresována. *Skeleton* nakonec volá metodu daného *objektu* a vrací výsledek zpracování *orbu*, který jej opět přeloží do formy *IIOP* a odešle zpět klientovi.

2.2.1 Objektová reference

Než však může klient odeslat požadavek na server, musí nejprve získat *objektovou referenci*. *Objektová reference* v corbě zastupuje *objekt* serveru, kterému jsou adresovány požadavky a to nezávisle na lokalizaci objektu v síti. *Objektová reference* tedy může zastupovat jak objekt lokální, tak vzdálený, který se nachází na jiném hostitelském počítači. *Objektová reference* je v corbě definována jako objekt, který implementuje rozhraní *CORBA::Object*. Toto rozhraní kromě jiného deklaruje metody pro dynamickou identifikaci rozhraní objektu a metody které umožňují vytvořit klientovi požadavek. Corba rovněž definuje alternativní reprezentaci *objektové reference* v různých formátech umožňující její transport a výměnu mezi výpočetními systémy v elektronické i neelektronické podobě. V textové podobě je možné reprezentovat objektovou referenci ve formátu *IOR (Interoperable Object Reference)* a ve formátech *corbaloc* a *corbaname*. Tyto formáty jsou kompletně popsány v [1] a [2].

2.3 Komponentový model

V corbě mají aplikace jednotnou základní architekturu (viz. obr. 3) a používají množinu v corbě definovaných komponent. Aplikace musí obsahovat všechny nezbytné komponenty, které potřebují ke své činnosti. Tyto komponenty by měly obsahovat všechny dostupné implementace corby. Těmito komponentami jsou *ORB* a *objektové služby (object services)*. Základní komponenty bývají integrální součástí aplikace, objektové služby jsou často řešeny jako samostatné komponenty, ke kterým mohou přistupovat klienti a server. Některé komponenty mohou v implementacích chybět, anebo mohou být řešeny v různých implementacích rozdílným způsobem, avšak v souladu s *OMG* normou.

2.3.1 ORB

Komponenta *ORB (Object Request Broker)* tvoří jádro komunikace v *corbě*. *Orb* je komponenta, která propojuje aplikace klienta a serveru v distribuovaném prostředí. Jednotlivé orby si vyměňují zprávy, požadavky a odpovědi (viz. odst. 2.2 Komunikační model). Klienti i servery mají zpravidla svého vlastního *orba*, který zajišťuje vlastní vysílání i příjem zpráv. *Orb* je také klíčem ke vzájemné součinnosti aplikací v distribuovaném prostředí. V distribuovaném prostředí mohou aplikace běžet na různých platformách a systémech, které mohou používat různý hardware i software, různé operační systémy. Aby bylo dosaženo této součinnosti aplikací, musí *orby* používat jednotný komunikační protokol. Pro jednotnou komunikaci mezi *orby* byl definován protokol *IIOP (Internet Inter Orb Protocol)*. Protokol IIOP je nadstavbou protokolu TCP/IP. Vývojář distribuované aplikace v corbě obvykle nevidí implementační detaily komunikace mezi *orby*. *Orb* však poskytuje aplikaci veřejné rozhraní. Klient i server má možnost volat metody rozhraní orbu za účelem iniciace odeslání anebo příjmu žádosti, ale také aby požádal o *objektové reference* na dostupné služby (jmenná služba, *IR*, a další, budou dále popsány). Server i klient má také k dispozici metody, kterými může kontrolovat programový běh vlastního *orbu*. Jelikož běžně v návrhu nepotřebujeme znát detaily vlastní implementace *orbu*, nepouštěl jsem se do jeho podrobné analýzy. Pro přehled a lepší pochopení základních konceptů corby jsem však v kapitole popsal některé pro návrh využitelné metody rozhraní *orbu*. Kapitola popisuje metody tzv. *jádra orbu* (*ORB Core*). Metody *jádra orbu* by měly být dostupné ve všech implementacích. Tyto metody jsou dostupné na straně klienta i serveru a je možné je rozdělit do několika skupin, viz. tab. 1.

Tab. 1: Některé užitečné metody rozhraní orbu

<table>
<tr><td rowspan="2">Metody pro inicializaci</td><td>ORB_init()</td></tr>
<tr><td>resolve_initial_references()</td></tr>
<tr><td rowspan="2">Metody informující o dostupných službách</td><td>list_initial_services()</td></tr>
<tr><td>get_service_information()</td></tr>
<tr><td rowspan="4">Metody pro komunikaci (DII)</td><td>send_multiple_requests_oneway()</td></tr>
<tr><td>send_multiple_requests_deferred()</td></tr>
<tr><td>poll_next_response()</td></tr>
<tr><td>get_next_response()</td></tr>
<tr><td rowspan="2">Metody pro konverzi objektové reference</td><td>object_to_string()</td></tr>
<tr><td>string_to_object()</td></tr>
<tr><td rowspan="5">Metody pro kontrolu vláken a programového běhu orbu</td><td>work_pending()</td></tr>
<tr><td>perform_work()</td></tr>
<tr><td>run()</td></tr>
<tr><td>shutdown()</td></tr>
<tr><td>destroy()</td></tr>
</table>

2.3.1.1 Metody pro inicializaci

Tato skupina zahrnuje běžně používané metody, ktreré umožňují inicializovat *orb* a získat *objektovou referenci orbu* a *objektových služeb*, které jsou v dané implementaci dostupné.

CORBA::ORB_init()
Metoda inicializuje implementaci *orbu* a vrací jeho referenci. Parametry zahrnují jednoznačný identifikátor typu *CORBA::ORBid*, který určuje implementaci *orbu*, která má být z dostupných použita pro vytvoření instance *orbu*, a parametr, který umožňuje předat sekvenci hodnot typu string, která obsahuje pro inicializaci specifické parametry jako *url* a *port* jmenné služby a podobně.

ORB::resolve_initial_references()
Metoda vrací *objektovou referenci* požadované *objektové služby*. *Objektová služba* je identifikována hodnotou typu *ObjectId*. V corbě mají standardní objektové služby tyto hodnoty rezervovány.

2.3.1.2 Metody informující o dostupných službách

ORB::list_initial_services()
Metoda vrací seznam hodnot typu *ObjectId*, které identifikují *objektové služby* dostupné voláním metody *ORB::resolve_initial_references()*.

ORB::get_service_information()
Této metody je možné využít ke zjištění, které služby a jejich rysy jsou v této implementaci *orbu* dostupné pro uživatele. Metoda přijímá jako parametr typ služby a vrací logickou hodnotu a výstupní parametr obsahující detailnější informace o službě. Hodnota *true* indikuje, že orb danou službu podporuje, hodnota *false* indikuje, že orb službu nepodporuje.

2.3.1.3 Metody pro komunikaci

Metody jsou popsány v odstavci *2.3.5.1 Rozhraní Dynamic Invocation Interface (DII)*.

2.3.1.4 Metody pro konverzi objektové reference

Tyto metody se používají pro konverzi objektové reference z řetězcového formátu na objekt typu *CORBA::Object* a naopak. Jako řetězce jsou nejčastěji ve formátu *IOR* (*Interoperable Object Reference*), *corbaloc* a *corbaname*.

ORB::string_to_object()
Metoda přijímá jako parametr objektovou referenci jako řetězec a vrací objektovou referenci jako objekt typu *CORBA::Object*.

ORB::object_to_string()
Metoda přijímá jako parametr objektovou referenci jako objekt typu CORBA::Object a vrací její řetězcovou podobu.

2.3.1.5 Metody pro kontrolu vláken a programového běhu orbu

Tyto metody umožní předat výpočetní zdroje jako jsou vlákna *orbu*. Orb využije získané vlákno a vykoná v něm potřebné interní operace. Některé implementace *orbu* potřebují navíc ke své činnosti vlákno *main*. V tom případě uživatel volá tyto metody ve vlákně *main*.

ORB::work_pending()
Tato metoda vrací logickou hodnotu, která informuje volající vlákno, zda *orb* potřebuje k vykonání interních operací využít vlákno *main*. Hodnota *true* informuje, že *orb* potřebuje ke své činnosti vlákno *main*. Hodnota *false* naopak říká, že *orb* k vykonání své činnosti vlákno *main* nepotřebuje.

ORB::perform_work()
Voláním funkce v kontextu vlákna *main* aplikace předá vlákno main *orbu*, který má tímto možnost v tomto vlákně vykonat potřebné operace. Poté *orb* vrací vlákno zpět aplikaci. Voláním metody z jiného vlákna než *main*, metoda neprovádí nic (jednoduše vykoná *return*).

ORB::run()
Aplikace voláním metody *run()* předá řízení *orbu*, který má tímto možnost vykonat potřebné interní operace. *Orb* předá řízení zpět poté, co je na něm volána metoda *ORB::shutdown()*. Některé *orby* potřebují ke své funkci vlákno *main*. V tom případě by měla aplikace volat metodu ve vlákně *main* anebo v tomto vlákně použít metody *work_pending()* a *perform_work()* v *polling* smyče.

ORB::shutdown()
Tato metoda informuje *orb* aby ukončil svou činnost za účelem plánovaného volání metody *destroy()*. Ukončení činnosti *orbu* způsobí, že budou také zdestruovány všechny objektové adaptéry, které bez *orbu* nejsou schopny samostatné existence. Metoda obsahuje parametr *wait_for_completion*. Je-li parametr *true*, metoda blokuje vlákno až do doby kdy skončí *shutdown* proces. Učiní-li tak aplikace ve vlákně, které obsluhuje volání, *orb* neukončí činnost a vyvolá vyjímku, protože blokování by mohlo vyústit v deadlock.

ORB:: destroy()
Tato metoda zdestruuje instanci *orbu* a uvolní všechny *orbem* alokované výpočetní zdroje. Jestliže dosud nebyla volána metoda *shutdown()* pak metoda *destroy()* zavolá nejprve tuto metodu s příznakem *wait_for_completion* nastaveným jako *true*.

Příklad *polling* smyčky, kdy *orb* využívá vlákno *main:*

```
// Java
while (true) {
  if (orb.work_pending() == true) {
    orb.perform_work();  // orb vykoná implementačně závislé operace
  }
  // vlákno main vykoná vlastní operace
  if (finished == true) break;
}
```

2.3.2 Objektový adaptér

Funkci *objektového adaptéru* již částečně vysvětluje jeho název. Objektové adaptéry tvoří spojovací článek mezi dvěma komponentami, *orbem* a *objektem* (viz. obr. 3). Smyslem objektového adaptéru je vyčlenit z implementace *orbu* operace, které manipulují s *objekty* a vykonávají jejich správu. Důvodem je, že *orb* by mohl s objekty

pracovat pro ně nevhodným způsobem. Implementace rozhraní objektového adaptéru je přizpůsobena a sladěna pro použití s adaptérem vázanými objekty. Objekty se tak mohou rozhodnout pro objektový adaptér, který jim nejlépe vyhovuje a mohou vybrat ten, který je pro ně nejlépe přizpůsoben. Objektové adaptéry *orbu* poskytují pouze rozhraní a *orb* tak pracuje s objekty transparentně. Na serveru může existovat jeden anebo i několik *objektových adaptérů*. Každý adaptér propojuje *orb* s určitou množinou objektů, přičemž obvykle pro danou množinu vykonává potřebnou správu a obvykle může uplatňovat různé politiky (policies). *Orb* s objektem zachází prostřednictvím *objektového adaptéru*, který je s tímto objektem asociován. Objekt také může využívat služby *orbu* prostřednictvím svého *objektového adaptéru*. *Objektový adaptér* poskytuje metody, které umožňují s adaptérem registrovat *objekty*, vytvářet *objektové reference* a zajišťuje mapování *objektů* na *objektové reference*. Smysl *objektového adaptéru* je také v implementaci služeb a metod, které nejsou součástí implementace *orbu* ani *objektových implementací*, ale které se těchto dvou komponent nějakým způsobem dotýkají. Tyto metody umožňují kontrolovat průběh interakce mezi *orbem* a *objekty* a zahrnují bezpečnostní mechanismy, pravidla pro životnost objektů na serveru, pravidla pro aktivaci a deaktivaci objektů apod. *Orb* vykonává pouze základní funkce, které umožňují průhlednou a jednoduchou komunikaci mezi různými *orby* a mezi *orby* a *objektovými adaptéry*. Ostatní funkce může implementovat *objektový adaptér*. V corbě je za účelem širšího použití deklarován konvenční objektový adaptér, který se označuje jako *POA (Portable Object Adapter)*.

2.3.2.1 Objektový adaptér POA (Portable Object Adapter)

Adaptér *POA* nahrazuje dřívější konvenční objektový adaptér *BOA (Basic Object Adapter)*, který byl z corby pro nedostatky vyčleněn. Cílem *POA* bylo navrhnou koncept mnohem univerzálnějšího objektového adaptéru, který by obecně pokrýval potřebu většiny *objektových implementací* a který by byl použitelný a široce používaný ve většině dostupných implementací *corby* a zajistil tak mezi nimi portabilitu objektových implementací. Adaptér *POA* zahrnuje řadu rysů a konceptů, které umožní přizpůsobit adaptér objektům. V této kapitole budou zmíněny pouze ty nejdůležitější, ostatní je možné nalézt ve specifikaci corby.

2.3.2.2 Hierarchie POA adaptérů, vytváření POA

Adaptéry *POA* mohou vytvářet uspořádanou hierarchii, která umožňuje objekty rozdělit do samostatných množin. Kořenem této hierarchie je tzv. *kořenový POA adaptér (Root POA)*. Ostatní adaptéry v hierarchii jsou od něj odvozeny a označují se jako *dceřinné POA adaptéry*. Dceřinné adaptéry je možné různě konfigurovat podle potřeby s nimi asociovaných objektů. Umožňují tak zcela odlišné zpracování žádostí a aktivace objektů, odlišné životní cykly objektů, různou úroveň perzistence objektů a podobně. Pro vytváření dceřinných *POA* adaptérů slouží metoda *POA::create_POA()*. Metoda dovoluje specifikovat jméno, tzv. POA manager (viz. *2.3.2.11 POA manager*) a různé politiky vytvářeného adaptéru (viz. *2.3.2.3 POA politiky*).

2.3.2.3 POA politiky

Jak již bylo zmíněno, *adaptéry POA* umožňují přiřadit různá pravidla co se týče vlastního zacházení s objekty. Tato pravidla definují tzv. *POA* politiky (*POA policies*). Politiky jsou objekty se společným předkem typu *CORBA::Policy*. Tyto objekty je

možné asociovat s *POA* a adaptér pak uplatňuje danou politiku na všechny vázané objekty. *POA politiky* můžeme vytvářet voláním metod *POA::create_<policy type>_policy()* jež vrací objekty *POA::<policy type>Policy*. Tyto objekty můžeme asociovat s *POA* při jeho vytváření metodou *POA::create_POA()*. V kapitole bude nyní následovat podrobný popis používaných *POA politik*. V současnosti je definováno několik *POA politik*, jenž jsou uvedeny v tab. 2.

Tab. 2 POA politiky

Servant Retention Policy
Request Processing Policy
Implicit Activation Policy
Thread Policy
Lifespan Policy
Id Assignment Policy
Object Id Uniqueness Policy

2.3.2.4 Vyhledání Servanta

Při doručení žádosti od klienta, *POA* nejprve vyhledá tzv. *servanta*. Servant je synonymní výraz pro instanci *objektu* - *objektové implementace*, která implmentuje služby serveru a která se použije ke zpracování žádosti. Při vyhledávání *servanta* v zásadě mohou nastat tři různé scénáře. *POA* může použít buď *aktivního servanta* nebo *implicitního servanta* (*default servant*) anebo *POA servanta* získá od *servant manageru*. Který scénář se uplatní závisí na nastavení politik *Servant Retention Policy* a *Request Processing Policy*.

2.3.2.4.1 Mapa aktivních objektů (AOM)

Pro udržování *aktivních servantů* používá objektový adaptér tzv. *mapu aktivních objektů* (zkráceně *AOM - Active Object Map*). V *AOM* jsou servanti registrováni s jejich *ObjectId*, hodnotou, která jednoznačně určuje servanta v *AOM* a podle které adaptér *POA* vyhledává servanta v *AOM*. Servant může být registrován s jedinou anebo i více hodnotami *ObjectId*, viz. *2.3.2.9 Politika Object Id Uniqueness Policy*. Tyto hodnoty *ObjectId* může alokovat sama aplikace anebo *POA adaptér*, viz. *2.3.2.8 Politika Id Assignment Policy*. Hodnota *ObjectId* může být adaptérem zapouzdřena do *objektové reference*. Klient tak může vytvořit požadavek (činí tak s pomocí *objektové reference*), který bude směrován na servanta s tímto *ObjectId*. Explicitně je možné vytvořit referenci voláním metod *POA::create_reference()* a *POA::create_reference_with_id()*. První metoda předpokládá nastavení politiky *Id Assignment Policy* s hodnotou SYSTEM_ID, druhá s hodnotou USER_ID.

2.3.2.4.2 Politika Servant Retention Policy

Implicitní hodnota politiky *Servant Retention Policy* je hodnota RETAIN. Může být nastavena ještě jedna hodnota NON_RETAIN. Hodnota RETAIN určuje, že *POA* má udržovat *servanty* aktivní, tj. v *AOM*. Hodnota NON_RETAIN naopak říká, že *POA* nemá udržovat *servanty* aktivní.

2.3.2.4.3 Politika Request processing Policy

Tato politika určuje způsob vyhledání *servanta* za účelem zpracování požadavku. Přitom se uplatňuje také hodnota politiky *Servant Retention Policy*, jak bylo popsáno výše.

1) Je nastaveno RETAIN a USE_ACTIVE_OBJECT_MAP_ONLY
Tento způsob lokalizuje servanta pouze s pomocí mapy *AOM*, která obsahuje pouze aktivní servanty. Jestliže v *AOM* servant s odpovídajícím *ObjectId* neexistuje, server vrací klientovi výjimku OBJECT_NOT_EXIST.

2) Je nastaveno USE_DEFAULT_SERVANT a NON_RETAIN
V tomto případě *POA* nehledá servanta v *AOM* a pro zpracování žádosti se použije vždy implicitní servant. Ten se nastaví voláním metody POA::set_servant(). Jestliže k *POA* adaptéru nebyl asociován *implicitní servant*, při vyřizování žádosti je klientovi vrácena zpět výjimka OBJ_ADAPTER.

3) Je nastaveno USE_DEFAULT_SERVANT a RETAIN
Jestliže je nastaveno RETAIN, adaptér *POA* se nejprve pokusí servanta lokalizovat v *AOM* a to podle daného *ObjectId*. Jestliže takový servant v *AOM* neexistuje, pro zpracování žádosti se použije implicitní servant. Jestliže k *POA* nebyl asociován *implicitní servant*, při vyřizování žádosti je klientovi vrácena zpět výjimka OBJ_ADAPTER.

4) Je nastaveno RETAIN a USE_SERVANT_MANAGER
K lokalizaci se použije *AOM* anebo i tzv. *servant manager*. V tomto případě, kdy je nastaveno RETAIN se použije *servant manager* typu *ServantActivator*. Adaptér se nejprve pokusí servanta s daným *ObjectId* lokalizovat v *AOM*. Jestliže servant není přítomen, adaptér použije *servant managera*. Může nastat případ *Aktivace servanta podle potřeby* (viz. Aktivace podle potřeby). Jestliže k *POA* nebyl asociován *servant manager*, klientu je vrácena zpět výjimka OBJ_ADAPTER. Servant managera je možné s adaptérem asociovat s pomocí metody *POA::set_servant_manager()*.

5) Je nastaveno NON_RETAIN a USE_SERVANT_MANAGER
V tomto případě *AOM* nemusí existovat a *POA* nehledá *servanta* v *AOM*. *POA* použije servant managera. V tomto případě, kdy je nastaveno NON_RETAIN se použije servant manager typu *ServantLocator* a *POA* volá metodu *ServantLocator::preinvoke*. Tato metoda vrací *servanta* jako návratovou hodnotu. Vrácený *servant* je použit pro zpracování žádosti. Jestliže k *POA* nebyl asociován *servant manager*, klientu je vrácena zpět výjimka OBJ_ADAPTER.

2.3.2.5 Aktivace Servanta

Pokud nebyl nastaven *implicitní servant* ani *servant manager*, pak *POA* pro lokalizaci může použít pouze *AOM*. V tom případě musí být servant nejprve aktivován, tj. zařazen do *AOM*. Aktivován může být buď explicitně anebo může za určitých okolností dojít k implicitní aktivaci anebo tzv. aktivaci podle potřeby.

2.3.2.5.1 Explicitní aktivace

Server může aktivovat *servanta* explicitně voláním metody *POA::activate_object()* anebo voláním metody *POA::activate_object_with_id()*.

2.3.2.5.2 Aktivace podle potřeby

Tento typ aktivace servanta se uplatňuje, když *POA* má nastaveny politiky RETAIN a USE_SERVANT_MANAGER. Aplikace k adaptéru *POA* nastavuje uživatelsky implementovaného *servant managera* typu *ServantActivator*. Aplikace nastaví k *POA* servant managera voláním metody *POA::set_servant_manager()*. *POA* použije *servant managera* kdykoli přijme žádost, pro kterou v *AOM* neexistuje odpovídající *ObjectId*. V tom případě *POA* volá metodu *ServantActivator*::*incarnate(),* která vytvoří *servanta* a vrací jej jako návratovou hodnotu. Protože je nastaveno RETAIN, *POA* asociuje servanta s požadovaným *ObjectId* v *AOM (*aktivuje *servanta)*.

2.3.2.5.3 Implicitní aktivace

Implicitní aktivace umožňuje automatickou aktivaci servanta bez potřeby jeho explicitní aktivace. K implicitní aktivaci může dojít automaticky při volání některých metod, které vykonávají akce, kterým by měla předcházet aktivace servanta a servant dosud není aktivní.
Implicitní aktivace vyžaduje současné nastavení SYSTEM_ID a RETAIN. Metody, které podporují implicitní aktivaci jsou také metody *POA::servant_to_reference()* a *POA::servant_to_id()*. Zda dojde k implicitní aktivaci servanta je možné určit nastavením politiky *Implicit Activation Policy*.

2.3.2.5.3.1 Politika Implicit Activation Policy

Politika umožňuje nastavit dvě hodnoty. Hodnota IMPLICIT_ACTIVATION říká, že *POA* bude používat mechanismu implicitní aktivace.
Hodnota NO_IMPLICIT_ACTIVATION tento mechanismus zakazuje a je potřeba objekty aktivovat explicitně. Hodnota NO_IMPLICIT_ACTIVATION je nastavena implicitně.

2.3.2.6 Deaktivace Servanta

Jestliže je servant aktivní, můžeme jej explicitně deaktivovat metodou *POA:: deactivate_object()*. Voláním této metody informujeme adaptér, aby deaktivoval tohoto servanta hned jak to bude možné, tj. po zpracování všech právě aktivních požadavků. Od okamžiku volání jsou požadavky, které jsou adresovány tomuto servantu označeny jako neaktivní a mohou vracet vyjímku, která informuje klienta, že servant není aktivní. Často se uplatňuje deaktivace objektů implicitně. Ta může nastat např. když *orb* anebo *POA* adaptér usoudí, že objekt musí být deaktivován. Může být například specifikováno, že objekt má být deaktivován po určitém časovém intervalu, kdy na server nepřišel žádný požadavek na tento objekt anebo počet aktivních objektů přesáhl určitý limit. Implicitně jsou také deaktivovány všechny aktivní objekty, když je volána metoda *POAManager::deactivate()*, tj. když *POA* přechází do stavu *inactive* (viz. *2.3.2.11 POA manager*).

2.3.2.7 Životnost Servanta

Životnost *servanta* lze ovlivnit politikou *LifespanPolicy*. Tato politika určuje, jestli může *servant* existovat i mimo kontext *POA*. Lze nastavit dvě hodnoty politik, hodnoty TRANSIENT a PERSISTENT. Implicitně je v *POA* nastavena politika TRANSIENT.

2.3.2.7.1 Politika LifespanPolicy

TRANSIENT - tato hodnota říká, že objekt nemůže existovat v neaktivním *POA* adaptéru. Jakmile se *POA manager* dostane do stavu *inactive* (viz. *2.3.2.11 POA manager*) všechny žádosti na tomto *POA adaptéru* způsobí návrat výjimky OBJECT_NOT_EXIST.

PERSISTENT - tato hodnota říká, že objekt může existovat i v rámci neaktivního *POA adaptéru*. Žádosti mohou vyvolat implicitní aktivaci *POA adaptéru* a *servanta*.

2.3.2.8 Politika Id Assignment Policy

Tato politika určuje způsob alokace hodnot *ObjectId* pro servanty. Politika USER_ID sděluje, že adaptér nemá alokovat hodnoty *ObjectId* pro *sevanty* a o alokaci se musíme postarat sami. Hodnota SYSTEM_ID naopak určuje, že o alokaci se postará *POA adaptér*.

2.3.2.9 Politika Object Id Uniqueness Policy

Politika umožňuje aktivovat *servanta* s jedinou anebo více hodnotami *ObjectId*. Politika UNIQUE_ID určuje, že *servant* podporuje pouze jednu hodnotu *ObjectId*. Pomocí politiky MULTIPLE_ID může mít *servant* i více *ObjectId*.

2.3.2.10 Management vláken v POA

Pro zpracování žádostí ve vícevláknovém systému může být použito více vláken současně. Vláknový model, který se při zpracování použije je možné určit nastavením politiky *ThreadPolicy*.

2.3.2.10.1 Politika ThreadPolicy

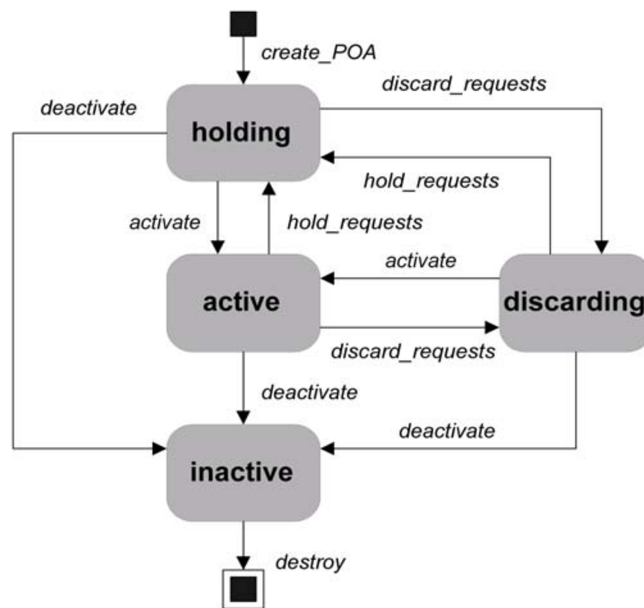
ORB_CTRL_MODEL - v tomto modelu mohou být žádosti zpracovávány souběžně (paralelně). Má-li *POA* přiřazenu tuto politiku, může vytvářet více vláken za účelem zpracování žádostí. Zpravidla až do určitého limitu, který je dán implementací. Obvykle se použije jedno vlákno pro každou žádost. POA přitom může využít tzv. zásobník vláken, do kterého ukládá jednou vytvořená a aktuálně nevyužitá vlákna. Poté co přijde od klienta požadavek, může *POA* z tohoto zásobníku přidělit žádosti vlákno. Velikost zásobníku může být za běhu optimalizována. Politika ORB_CTRL_MODEL je pro adaptéry výchozí.

SINGLE_THREAD_MODEL - v tomto modelu *POA* používá pro zpracování všech požadavků jedno vlákno a všechny žádosti pro objekty jsou na tomto adaptéru zpracovávány sekvenčně.

MAIN_THREAD_MODEL - v tomto modelu *POA* využívá vlákno "main" pro všechny žádosti. Použije se tehdy, když je potřeba, aby *servant* zpracovával žádost výhradně ve vlákně main. Žádosti jsou zpracovávány sekvenčně.

2.3.2.11 POA manager

S každým *POA adaptérem* je asociován tzv. *POA manager*. *POA manager* reprezentuje procesní stavy *POA*, tj. stavy, ve kterých *POA* zastává různé aktivity vůči požadavkům. Voláním metod *POA manageru* je možné tyto aktivity explicitně ovlivňovat. *POA* může zastávat čtyři procesní stavy *holding*, *active*, *discarding* a *inactive*. Tyto stavy mohou být znázorněny ve stavovém diagramu, viz. obr. 4.



Obr. 4: Stavový diagram POA manageru

Procesní stavy POA manageru

holding
V tomto stavu *POA* přijímá žádosti, ale nezpracovává je. Příchozí žádosti jsou ukládány do fronty. Žádosti jsou přijímány až do dosažení limitu, který závisí na implementaci *POA*. Po překročení limitu jsou požadavky odmítnuty a klientu je doručena výjimka TRANSIENT, která klientu sděluje, že aktuálně nemůže být požadavek přijat a klient se může pokusit požadavek zaslat později.

active
V tomto stavu *POA* přijímá žádosti a jestliže jsou dostupné zdroje (mj. pracovní vlákna), pak také žádosti zpracovává. Jak jsou žádosti zpracovávány, závisí na modelu práce vláken (viz. Management vláken v *POA*). Příchozí žádosti jsou ukládány do fronty až po dosažení limitu. Po překročení limitu jsou požadavky odmítnuty a klientu je doručena výjimka TRANSIENT, která klientu sděluje, že aktuálně nemůže být požadavek přijat a klient se může pokusit požadavek zaslat později.

discarding
V tomto stavu *POA* odmítne všechny příchozí požadavky. *POA* pro každou odmítnutou žádost klientovi vrací výjimku TRANSIENT. Klient se může pokusit požadavek později zopakovat.

inactive
V tomto stavu *POA* odmítne všechny nově příchozí požadavky a zahodí také všechny již došlé požadavky, které *POA* dosud nezačal zpracovávat. V tomto stavu se *POA* může ocitnout voláním metody *deactivate().*

2.3.2.12 Aktivace POA adaptéru

S adaptérem *POA* je možné asociovat také tzv. *aktivátor adaptéru*. *Aktivátor adaptéru* nabízí možnost *runtime* vytvoření *dceřinných POA adaptérů*, pro které došel na server požadavek a které dosud nebyly vytvořeny anebo byla volána metoda *POA:: find_POA()*, které byl předán parametr *activate=TRUE*. Pro vytvoření dceřinných adaptérů je potřeba, aby předchůdce měl asociovaného *aktivátora adaptéru*. Jestliže chybí, pak *orb* klientovi vrátí výjimku OBJECT_NOT_EXIST. Jestliže je *aktivátor adaptéru* přítomen, je na něm volána metoda *unknown_adapter()*, která je odpovědná za vytvoření *dceřinného POA adaptéru*.

2.3.3 Jmenná služba (Naming Service, NS)

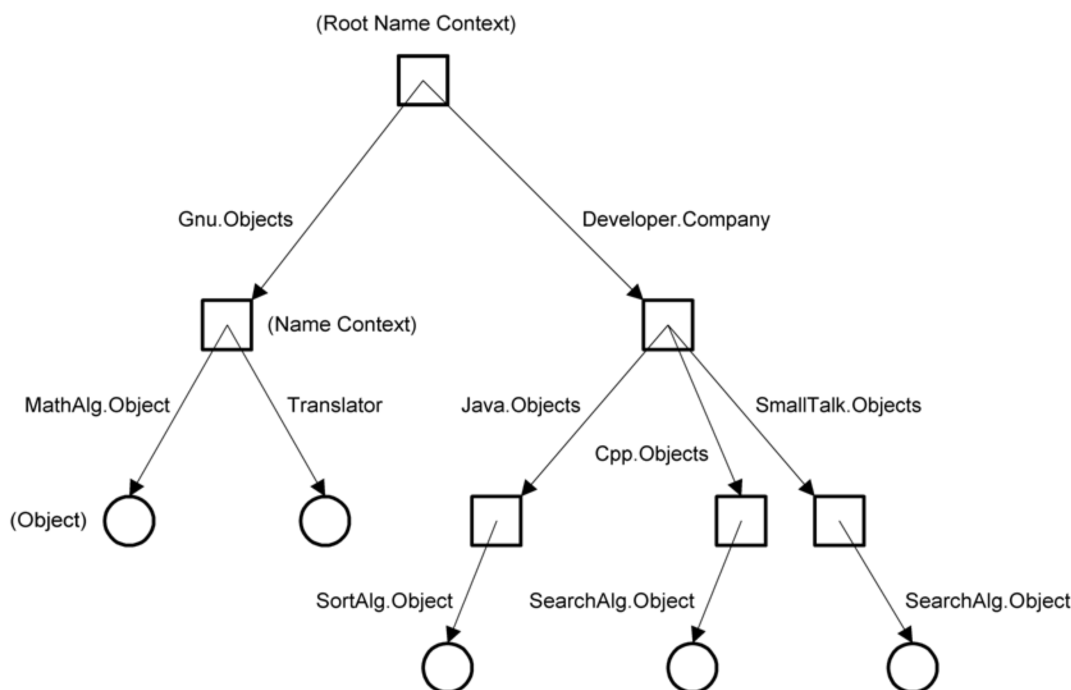
Jmenná služba slouží pro podporu evidence, vyhledávání a lokalizace objektových implementací v síti. Jmenná služba vytváří hierarchický jmenný prostor, ve kterém udržuje údaje o umístění objektů v síti. Jmenná služba umožňuje serveru registrovat objektové reference se jmény. Klienti mohou listovat ve jmenné službě a podle jména získat objektovou referenci na nějaký objekt. Pro snazší terminologii, v této kapitole bude pojem objekt chápán také ve smyslu objektové reference. V této kapitole budou vysvětleny pouze běžné koncepty jmenné služby a popis jejího rozhraní. Pro detailní informace o jmenné službě odkazuji na specifikaci [2]. Rozhraní jmenné služby je poměrně jednoduché a umožňuje klientům jednoduchý ale efektivní přístup k objektům.

2.3.3.1 Jmenný prostor v NS

Prostor jmen v *NS* tvoří hierarchickou strukturu podobnou stromu, viz. obr. 5. V něm se nacházejí kořen, uzly, větve a listy. Uzly představují tzv. *jmenné kontexty*. Kořen představuje *kořenový jmenný kontext*. Listy představují objekty vázané ve jmenné službě. Větve nesou jména a pojmenovávají buď jmenné kontexty, jestliže větev vede do běžného uzlu, anebo pojmenovávají objekty, jestliže větev vede do listu.

2.3.3.2 Rozhraní jmenné služby

Jmenná služba obsahuje pouze jediný modul *CosNaming*, viz. tab. 3. Tento modul obsahuje tři struktury a tři rozhraní s metodami, viz. tab. 4. V následujících odstavcích bude uveden jejich popis.



Obr. 5: Schématické vyjádření jmenného prostoru v NS

Tab.3: Modul CosNaming

Modul	Struktury	Rozhraní
CosNaming	NameComponent Name Binding	NamingContext BindingIterator NamingContextEx

Tab.4: Rozhraní jmenné služby

Rozhraní	Operace	Výjimky
NamingContext	bind() rebind() bind_context() rebind_context() bind_new_context()	InvalidName AlreadyBound NotFound NotEmpty CannotProceed
	unbind()	
	new_context()	
	resolve()	
	list()	
	destroy()	
BindingIterator	next_one() next_n()	
	destroy()	
NamingContextEx	k nalezení ve specifikaci [2]	

2.3.3.3 Jména, jmenné vazby, jmenné kontexty

Jména jsou v *NS* strukturovaná a tvoří složené objekty typu *CosNaming::Name*. *Jméno* není pouze řetězec, každé *jméno* tvoří sekvenci *jmenných komponent*. *Jmenné*

komponenty jsou v *NS* objekty typu *CosNaming::NameComponent*. *Jméno* obsahující právě jednu *komponentu* se označuje jako *jednoduché jméno*, *jméno* složené z více *komponent* jako *složené jméno*.

Jednoduchá jména pojmenovávají k nim vázané *objekty* nebo vázané *jmenné kontexty*. Tato asociace *jména* a *objektu* se v *NS* označuje jako *jmenná vazba*. *Jmenná vazba* je objekt typu *CosNaming::Binding*. Používají se v operacích vázání objektů (*bind, bind_context*).

Složená jména pojmenovávají *objekty*, včetně nadřazených *jmenných kontextů*, tedy určují také jednoznačně cestu k *objektu* ve jmenném prostoru. Každá komponenta ve *složeném jméně* s výjimkou poslední komponenty slouží k pojmenování *jmenného kontextu*, poslední komponenta pojmenovává ke jménu vázaný *objekt*. Používají se v operacích vyhodnocování , viz. např. *resolve()*.

Jmenný kontext je *objekt*, který obsahuje množinu *jmenných vazeb*, tedy množinu dvojic *jmen* a *objektů*. *Jména* musí být v rámci *jmenného kontextu* unikátní.

2.3.3.4 Řetězcová jména

Řetězcová jména mají jednoznačně reprezentovat objekty typu CosNaming::Name ve tvaru řetězce a mají jednoduchou syntaxi. S *řetězcovými jmény* dovolují pracovat některé operace v *NS*. Syntaxe rezervuje tři významové znaky '/', '.' a '\'. Znak '/' ve *jméně* odděluje *jmenné komponenty*. Znak '.' odděluje složky *id* (pojmenovávající složka) a *kind* (významová složka) *jmenné komponenty*. Znak '\' slouží ke změně (neutralizaci) významu znaku '.' a '/' a '\'. Syntaxi je možné si ukázat na příkladech.

Tab. 5: Příklady vyhodnocení řetězcových jmen

Zápis	*Vyhodnocení*					
Řetězcové jméno	*Jmenné komponenty*					
	id	*kind*	*id*	*kind*	*id*	*kind*
"Translator"	Translator	<empty>				
"SortAlg.Object"	SortAlg	Object				
"Gnu.Objects/Translator"	Gnu	Objects	Translator	<empty>		
"a\/b\\c\.d/e\.f.g/h"	a/b\c.d	<empty>	e.f	g	h	<empty>
"a./.d/."	a	<empty>	<empty>	d	<empty>	<empty>

2.3.3.5 Služby NS

2.3.3.5.1 Vytvoření jmenného kontextu

NS umožňuje vytvořit jmenný kontext dvěma metodami, *new_context()* a *bind_new_context()*. Operace *new_context()* nedělá nic jiného, než že vrací nově vytvořený jmenný kontext. Operace *bind_new_context()* vytváří nový jmenný kontext a ve jmenném kontextu, nad kterým je metoda volána, vytváří jmennou vazbu se jménem jako parametrem metody.

2.3.3.5.2 Rušení jmenného kontextu

Rušení jmenného kontextu zajišťuje metoda *delete()*. Jestliže jmenný kontext obsahuje jmenné vazby, metoda způsobí výjimku *NamingContext::NotEmpty*.

2.3.3.5.3 Listování jmenným kontextem

Operace listování umožňuje procházet jmenné vazby vázané ve jmenném kontextu. *NS* podporuje operaci listování metodou *NamingContext::list()*. Metoda přijímá parametr udávající požadovaný maximální počet jmenných vazeb ve výstupním seznamu (*CosNaming::BindingList*). Výstupem je seznam jmenných vazeb a iterátor (*CosNaming::BindingIterator*), kterým jsou dostupné zbývající jmenné vazby ve jmenném kontextu.

2.3.3.5.4 Vázání objektů

Operace vázání umožňuje se jménem asociovat objekt nebo i jmenný kontext. Poté, co je s daným *objektem* asociováno *jméno*, je možné ho získat metodou *resolve()*. *NS* podporuje operace vázání *bind()*, *bind_context()*, *rebind()*, *rebind_context()* a také *bind_new_context()*. Operace *bind* umožňují vázat *objekt* (*jmenný kontext*) ke *jménu* pouze jednou, opakované volání způsobí výjimku *NamingContext::AlreadyBound*. Operace *rebind* umožňuje nahradit objekt již jednou vázaný.

2.3.3.5.5 Odvázání objektů

Operace odvázání umožňuje ze *jmenného kontextu* odstranit *jmennou vazbu*. *NS* podporuje operaci odvázání *unbind()*. Operace vázaný objekt nevrací.

2.3.3.5.6 Získávání objektů

Operace umožňuje získat zpět objekt vázaný k danému *jménu* ve *jmenném kontextu*. *NS* podporuje tuto operaci metodou *resolve()*. Parametrem metody je *jméno*, které musí jednoznačně určovat objekt včetně cesty. Za přetypování na původní typ je odpovědný klient *NS*.

2.3.3.5.7 Rozhraní BindingIterator

Toto rozhraní zapouzdřuje metody, které umožňují listovat jmenné vazby. K listování poskytuje operace *next_one()* a *next_n()*. Ke zdestruování objektu typu BindingIterator dává operaci *destroy()*, kterou je potřeba volat po použití iterátoru.

2.3.4 Statická komunikace klient - server

V tomto modelu komunikace se předpokládá, že klient má k dispozici typové informace o vzdálených objektech a zná jejich rozhraní již v době překladu. V tomto modelu spolu komunikuje klient a server předem definovaným způsobem. Tento model neumožňuje přidávat za běhu aplikace nové objekty a také neumožňuje dynamicky za běhu vytvářet nové požadavky. Tento model komunikace se použije tehdy, když bude architektura distribuované aplikace dána předem se všemi objekty a nebude se dále za běhu rozšiřovat.

2.3.4.1 IDL Stub

V tomto modelu klient zasílá žádosti na server přes tzv. *IDL Stub*. *IDL Stub* je objekt, který klientovi zpřístupňuje rozhraní služeb vzdáleného objektu, a to tak, že

implementuje identické rozhraní jako vzdálený objekt. Neimplementuje však služby jako takové, ale pouze konektivitu, která deleguje žádosti o služby na server. Protože *IDL Stub* je závislý na rozhraní vzdáleného objektu, klient vlastní několik *IDL Stubů* pro všechny typy objektů. Klient volá metody *IDL Stubu* syntakticky stejně jako by objekt implementující služby existoval v kontextu klienta. Ve skutečnosti *IDL Stub* předává žádosti *orbu* klienta a ten žádosti odesílá *orbu* serveru.

Rozhraní pro *IDL stub* a *objekty* se deklaruje v jazyce *OMG IDL*. Poté se *IDL* rozhraní přeloží překladačem do implementujícího jazyka. Překladač poskytne základní třídy pro *ILD Stub* a *objektovou implementaci* a po překladu tyto základní třídy již obsahují detaily komunikace a implementující třída objektu zdědí základní třídu a přidá chybějící implementaci. V kapitole o IDL jsem se pokusil shrnout základní a nejdůležitější rysy tohoto jazyka.

IDL Stub kromě (společného) *rozhraní IDL* implementuje rozhraní ObjectImpl. Stub používá metody rozhraní ObjectImpl aby předal požadavek *orbu* a následně od něj získal odpověď. Jakmile *stub* obdrží volání metody *IDL*, zachytí parametry předávané klientem a volá metodu ObjectImpl::_request(), která umožňuje sestavit požadavek. *Stub* následně požadavek předá *orbu* voláním metody ObjectImpl::_invoke() a *orb* požadavek přeloží do formátu *IIOP* a odešle na server. Poté co *orb* přijme a přeloží odpověď, metoda _invoke() ji předá *stubu*.

2.3.4.2 IDL Skeleton

Poté co *orb* na serveru přijme žádost od klienta, vyhledá s pomocí objektového adaptéru *objekt* a předá *objektu* žádost. To se děje prostřednictvím tzv. *skeletonu*. *Objekt* implementující služby kromě služeb implementuje také rozhraní *skeletonu*, přičemž se může rozhodnout, zda implementuje *IDL Skeleton* anebo tzv. *dynamický skeleton*. V tomto odstavci se předpokládá, že objekt implementuje *IDL Skeleton*. *Dynamický Skeleton* bude popsán v odstavci o dynamickém modelu komunikace. *Skeleton* plní obdobnou funkci jako *Stub* na straně klienta. Také *IDL Skeleton* je závislý na rozhraní objektu, tj. každý *objekt* implementuje vlastní *IDL Skeleton*.

Objekt implementující *IDL Skeleton* kromě (společného) *IDL rozhraní* implementuje rozhraní InvokeHandler. Metody tohoto rozhraní volá *orb*, aby objektu předal žádost od klienta. *Orb* předává žádost objektu metodou InvokeHandler::_invoke(). *Skeleton* po ověření správnosti požadavku a typové kontrole volá příslušnou metodu objektu, jenž implementuje zpracování požadavku. Parametrem metody _invoke() je buffer, z kterého si skeleton přečte údaje o požadavku (jméno metody, parametry a příznaky) a objekt typu ResponseHandler, který *skeleton* použije pro zapsání odpovědi pro klienta.

2.3.5 Dynamická komunikace klient - server

Tento model umožňuje za běhu aplikace vytvářet požadavky dynamicky a zároveň umožňuje za běhu přidávat nové objekty distribuované aplikace a umožňuje tak za běhu rozšiřovat server o nové služby. Rozhraní vzdálených objektů nemusí být známa v době překladu, tento druh komunikace dává k dispozici mechanismus jak identifikovat nové objekty a jejich rozhraní za běhu aplikace. Tento model komunikace se použije, bude-li se infrastruktura distribuovaných objektů dynamicky měnit anebo bude-li potřeba požadavky vytvářet dynamicky za běhu aplikace.

V tomto modelu klient zasílá požadavky na server s pomocí rozhraní *DII (Dynamic Invocation Interface)*. Poté co *orb* na straně serveru přijme žádost od klienta,

vyhledá objekt, a to stejným způsobem jako v případě statické komunikace. Namísto *IDL Skeletonu* se však použije tzv. *dynamický skeleton*.

2.3.5.1 Rozhraní Dynamic Invocation Interface (DII)

DII umožňuje klientovi vytvářet a zasílat žádosti pro objektové implementace dynamicky. Rozhraní *DII* se použije tehdy, když klientovi v době překladu ještě nejsou známy typové informace o objektech. Dynamická komunikace proto musí poskytovat mechanismus, který umožní klientovi rozpoznat objekty a jejich rozhraní za běhu programu (viz. *Interface Repository*). Tento způsob již není tak transparentní jako při použití *Stubu* a klient se musí sám postarat o sestavení žádosti a její zaslání s využitím rozhraní *DII*. *DII* nevyužívá *Stub*.

2.3.5.1.1 Sestavení požadavku

V *DII* se pracuje s objekty typu *Request*. Objekt *Request* reprezentuje žádost klienta neboli volání jedné metody na jednom *objektu* (*objektové implementaci*). Pro volání více metod je potřeba více objektů typu *Request*. Pro to, abychom mohli vytvořit *Request* objekt, je potřeba nejprve získat *objektovou referenci* pro *objekt*, kterému bude žádost adresována.

Poté co jsme získali *objektovou referenci* (v corbě *objektová reference* implementuje rozhraní CORBA::Object) je možné vytvořit *Request* voláním metody Object::_request(), které předáme *IDL* jméno operace anebo voláním Object::create_request(), která navíc umožňuje specifikovat seznam argumentů operace. Navíc je potřeba předat také parametr *result*, který reprezentuje návratovou hodnotu a do něhož bude zabalena odpověď. Metody vrátí objekt *Request*. Argumenty je možné přidat také opakovaným voláním metody *Request::add_arg()*. Operace *_request()* a *create_request()* dědí *objektová reference* od základního rozhraní CORBA::Object.

2.3.5.1.2 Odeslání požadavku

Požadavek zašleme na server použitím metod rozhraní *Request* anebo přímo prostřednictvím rozhraní *orbu*. Operace *Request::invoke()* předá požadavek *orbu*, který jej zašle na server. Metoda blokuje volající vlákno až do té doby, než přijde odpověď. Odpověď je zabalena do argumentu *result*, který byl specifikován při vytváření *Requestu*. Chce-li klient zaslat požadavek a pokračovat ve své činnosti během zpracování požadavku, namísto toho volá metodu *Request::send_deferred()*, která volající vlákno neblokuje. Pokud klient nepožaduje odpověď je možné použít i metodu *Request::send_oneway()*, která také neblokuje. Klient může navíc volat také několik požadavků současně neblokujícími metodami *ORB::send_multiple_requests_deferred()* a *ORB::send_multiple_requests_oneway()*.

2.3.5.1.3 Čekání na odpověď a příjem odpovědi

Pokud jsme k zaslání požadavku použili metodu *send*, můžeme se nyní dotazovat, zda již byl nastaven parametr *result* metodou *Request::poll_response()*. Metoda vrací booleovskou hodnotu. Metoda *Request::get_response()* nic nevrací, ale blokuje volající vlákno, až bude návratová hodnota v argumentu *result* nastavena. Obdobně pracují metody *ORB::poll_next_response()* a *ORB::get_next_response()*.

2.3.5.1.4 Destrukce požadavku

Nakonec, jestliže je zpracování požadavku u konce a klient získal odpověď, je potřeba objekt *Request* uvolnit voláním metody *Request::delete()*.

2.3.5.2 Rozhraní Dynamic skeleton Interface (DSI)

V dynamickém modelu komunikace se uplatňuje jiný princip volání metod objektu než ve statickém modelu. Objekt namísto *IDL skeletonu* implementuje tzv. *dynamický skeleton*. *Dynamický skeleton* umožňuje volat metody objektů, které klientovi v době překladu ještě nejsou známy - není znám typ ani rozhraní objektu. Objekty mohou být registrovány a aktivovány na serveru až v době provádění.

2.3.5.2.1 Předání požadavku objektu - rutina DIR

Předání požadavku klienta objektu se děje jednotným způsobem. *Orb* předá požadavek tzv. *dynamické implementační rutině* (zkráceně *DIR*), kterou implementuje každý objekt implementující *dynamický skeleton*. V této rutině *skeleton* zpracuje všechny nutné operace pro vlastní volání, tj. ověří správnost požadavku a provede typovou kontrolu a nakonec volá metodu implementující zpracování požadavku. *Skeleton* by měl ověřit, že daný objekt a metoda skutečně existuje. K tomu může požít rozhraní *IR (Interface Repository)* a princip dynamické identifikace.
V corbě *dynamický skeleton* implementuje rozhraní *DynamicImplementation* a jako *DIR* metodu *DynamicImplementation::invoke()*. Této metodě *orb* předá požadavek jako objektu typu *CORBA::ServerRequest*. Objekt *ServerRequest* má podobný význam jako objekt *Request* na straně klienta a obaluje všechna data potřebná pro samotné volání metody (jméno operace, argumenty, příznaky).

Získání jména operace a argumentů
DIR implementační rutina nejprve potřebuje identifikovat volanou metodu. Jméno operace si může přečíst z atributu *ServerRequest::operation*. Poté musí *DIR* volat metodu *ServerRequest::arguments()* čímž získá přístup k argumentům volané metody.

2.3.5.2.2 Zpracování požadavku klienta

Poté co *DIR* získá jméno operace a argumenty volání, ověří že objekt a metoda na serveru skutečně existuje a zkontroluje také argumenty volání. Kontrolu může provést s pomocí *IR* (viz. *2.3.5.3 Služba Interface Repository*). Následně může volat implementující metodu, které předá zjištěné parametry. Pokud během zpracování operace dojde k výjimce, *DIR* tuto zachytí a měla by ji vrátit zpět klientovi, aby ho informovala o neúspěšném zpracování. Výjimku pro klienta nastaví metodou S*erverRequest::set_exception()*. Pokud k výjimce při zpracování nedošlo, *DIR* může nastavit pro klienta výsledek metodou *ServerRequest::set_result()*. Poté *DIR* předá řízení zpět volajícímu *orbu*.

2.3.5.3 Služba Interface Repository (IR)

V distribuovaném systému může nastat situace, kdy server přidává nové objekty a služby až v době provádění. Aby klient mohl takové objekty rozpoznat a aby mohl identifikovat jejich rozhraní (služby), musí existovat způsob, kterým server tyto objekty

zveřejní (publikuje) a kterým zveřejní také rozhraní těchto objektů, včetně údajů jako parametry metod, které může klient předat apod. Tento mechanismus v *corbě* zastává *IR*.

IR je runtime služba či komponenta, která udržuje a zpřístupňuje typové informace o objektech. Výstižně lze *IR* charakterizovat tvrzením, že *IR* obsahuje rovnocenné údaje jako *IDL* soubory. Mohou zde být objekty jako modul (module), rozhraní (interface), metoda, parametr apod. Tyto objekty reprezentují jednotlivé *IDL* deklarace.

IR je za běhu přístupná jak klientovi tak na straně serveru. Proto je možné využít *IR* při dynamické komunikaci v corbě. Může však být využita mnoha dalšími způsoby. *IR* může například pomáhat při typové kontrole při volání metod objektů, protože k tomu má k dispozici všechny potřebné údaje. Server tedy může registrovat za běhu nové *objekty* (*objektové implementace*) a v *IR* o něm zapsat typové údaje a klient může k *IR* přistupovat a zjišťovat pro něj užitečné informace, např. zda je na serveru nový objekt, jakého je typu a jaké rozhraní a operace podporuje.

2.3.5.3.1 Struktura IR

IR tvoří hierarchickou strukturu objektů. Objekty představují jednotlivé deklarace nebo také obory platnosti (scope). Obory platnosti jsou objekty, jež implementují *IR* rozhraní *CORBA::Container*. Deklarace jsou zařazeny v rámci určitého oboru platnosti a implementují rozhraní *CORBA::Contained*. Kořenovým objektem (oborem platnosti) v *IR* je objekt implementující rozhraní *CORBA::Repository*, který poskytuje globální přístup k *IR*.

K tomu, abychom mohli pracovat s *IR* potřebujeme nejprve získat *objektovou referenci IR*. Tuto referenci je možné získat podoně jako pro ostatní služby voláním metody *orbu* ORB::resolve_intitial_references() a předáním identifikátoru *IR* (řetězec "InterfaceRepository"). Metoda vrací referenci typu *CORBA::Object*, proto je nutné pro další práci referenci přetypovat na *CORBA::Repository*.

2.3.5.3.2 Vyhledávání a čtení v IR

V *IR* můžeme vyhledávat objekty (deklarace) podle *nekvalifikovaného jména*, *kvalifikovaného jména* nebo *id*. Je to možné, protože každá deklarace (objekt typu *Contained*) má tyto tři atributy. *Jméno* i *id* jsou reprezentovány jako řetězce.

a) Nekvalifikované jméno
Nemusí být globálně jednoznačné, avšak musí být jednoznačné v rámci určitého oboru platnosti (scope). Obory platnosti jsou objekty typu *Repository*, *ModuleDef*, *InterfaceDef*, atd. Jednoduchá jména odpovídají identifikátorům v IDL souboru. Jednoduché jméno představuje typ *CORBA::Identifier*.

b) Kvalifikované jméno
Kvalifikované jméno je řetězec složený z nekvalifikovaných jmen, které jsou odděleny čtyřtečkou (::). Plně kvalifikované jméno navíc čtyřtečkou začíná a vyhodnocuje se relativně k oboru platnosti *Repository*. Každá deklarace (objekt typu *Contained*) má atribut pro plně *kvalifikované jméno*, které ji jednoznačně identifikuje v oboru *Repository*. Kvalifikované jméno představuje typ *CORBA::ScopedName*.

c) Id
Je to řetězec jednoznačně a globálně identifikující objekt (deklaraci) v objektu *Repository* a nemusí to být jméno. *Id* představuje typ *CORBA::RepositoryId*.

Pro vyhledávání a získávání obsahu *IR* jsou dostupné metody, deklarované v rozhraní *Container* jako *lookup()*, *lookup_name()* a *contents()* a v rozhraní *Repository* je dostupná metoda *lookup_id()*. Metody *lookup()* a *lookup_id()* přijímají jako parametry *kvalifikované jméno* a *id* a proto vrací jediný objekt typu *Contained* (deklaraci). Metoda *contents()* vrací sekvenci objektů *Contained* reprezentující všechny deklarace v rámci oboru platnosti. Metoda *lookup_name()* vrací rovněž sekvenci avšak podle *nekvalifikovaného jména*. Je možné specifikovat typ deklarace která se má vracet.

2.3.5.3.3 Identifikace typů a rozhraní pomocí IR

Metody určené pro získávání obsahu v *IR* vracejí typ *Contained*, proto je potřeba pro další práci s těmito objekty je nejprve přetypovat na správný typ. Ke zjištění, jakého jsou vrácené objekty vlastně typu, slouží metoda známá jako dynamická identifikace. Poté, když už je znám typ objektu, následuje vlastní přetypování.

V *IR* objekty (deklarace), vázané na typ, implementují rozhraní *CORBA::IDLType*. To obsahuje atribut *TypeCode*, který je v objektech přístupný pouze pro čtení. Objekt typu *TypeCode* obsahuje všechny potřebné informace o typu. Tento atribut se použije při *dynamické identifikaci typu*. Objekty implementující *IDLType* jsou objekty PrimitiveDef, TypedefDef, ValueDef, StringDef a podobně.

Pro *dynamickou identifikaci rozhraní* objektu je možno použít metodu *_get_interface_def()*, kterou vlastní každý objekt typu *CORBA::Object*. Tato metoda vrací objekt typu *CORBA::InterfaceDef*, což představuje v *IR* objekt rozhraní. Máme-li tedy k danému objektu *objektovou referenci* můžeme voláním této metody zpřístupnit definici rozhraní objektu.

2.3.5.3.4 Vytváření deklarací a zápis do IR

V *IR* je možné zapisovat objekty (deklarace) použitím metod *create_<type>*. Tyto metody vytvoří objekt daného typu (<type>Def) a zapíše jej v *IR*. Během provádění může nastat výjimka pokud uživatel zadal *jméno* nebo *id*, které koliduje s jiným nebo když deklarace v daném kontextu není povolena. Popis všech objektů <type>Def přesahuje možnosti textu, proto zájemce odkazuji na corba specifikaci [1], kde je možné nalézt detailní popis všech objektů.

2.4 Jazyk OMG IDL

V tomto odstavci bude uveden základní přehled gramatiky jazyka IDL. Tento úvod do jazyka IDL jsem se rozhodl začlenit proto, že v mnoha případech (když např. implementujeme aplikaci v souladu s pravidly uvedenými v odst. 2.3.4) se bez něj při implementaci aplikace v corbě neobejdeme. Ve své diplomové práci jsem znalostí tohoto jazyka využil, a v tomto jazyce deklaroval rozhraní pro CORBA-OPC můstek. Rozhraní je deklarováno v souboru *copc.idl* v adresáři *soft/src/copc/* na přiloženém CD-ROM. Detailní popis jazyka IDL je možné nalézt v [1].

2.4.1 Úvod do IDL

Jazyk OMG IDL (OMG Interface Definition Language) je čistě deklarativní jazyk. Slouží pro popis deklarací jako např. deklarací rozhraní tříd, deklarací metod a jejich parametrů, deklarací typů apod. To, že se jedná o deklarativní jazyk mimo jiné znamená, že v jazyce IDL nelze zapsat jazykové konstrukce jako řídící struktury apod. Jazyk IDL se snaží zachytit deklarativní rysy jiných jazyků (Java, C++, Smalltalk, ...) a postihnout je ve své syntaxi a tím je zobecnit a sjednotit. Díky tomu je možné aplikační rozhraní deklarovat obecně nezávisle na použitém implementačním jazyce. To umožnilo vytvořit pravidla pro mapování jazyka IDL pro specifické programovací jazyky, která jsou součástí specifikace IDL (pro jazyk C++ a Java viz. [8] a [9]). To s sebou přineslo možnost vytvořit překladače z jazyka IDL do těchto specifických jazyků. Jazyk IDL tím také usnadňuje vývoj aplikací určených pro distribuované prostředí, kdy je požadována jejich vzájemná součinnost nezávisle na použitém implementačním jazyce. Chceme-li např. implementovat server, je vhodné pro něj deklarovat rozhraní v jazyce IDL. Učiníme-li tak, pak nebude problém kdykoli implementovat klienta ať už jej budeme implementovat my sami nebo kdokoli jiný. Jednoduše se pak použije definice IDL rozhraní serveru a pro daný jazyk odpovídající IDL překladač (viz. také odst. 2.3.4). Nemusíme se pak starat jak který typ pro použitý jazyk serveru mapovat do použitého jazyka klienta, protože překladač tuto práci udělá za nás a automaticky vygeneruje rozhraní. My pak již jen jednoduše zdědíme rozhraní do třídy *stubu* klienta.

2.4.2 Přehled klíčových slov jazyka IDL, pravidla zápisu identifikátorů

Přehled klíčových slov jazyka je uveden v tab. 6.

Tab. 6: Přehled klíčových slov jazyka IDL

abstract	exception	inout	provides	truncatable
any	emits	interface	public	typedef
attribute	enum	local	publishes	typeid
boolean	eventtype	long	raises	typeprefix
case	factory	module	readonly	unsigned
char	FALSE	multiple	setraises	union
component	finder	native	sequence	uses
const	fixed	Object	short	ValueBase
consumes	float	octet	string	valuetype
context	getraises	oneway	struct	void
custom	home	out	supports	wchar
default	import	primarykey	switch	wstring
double	in	private	TRUE	

Klíčová slova nelze použít pro pojmenování identifikátoru, neboť jsou vyhrazena pro základní jazykové konstrukce. Při pojmenování identifikátoru klíčovým slovem hlásí překladač chybu. Jazyk *OMG IDL* není '*case sensitive*' jazyk. Překladač tedy hlásí chybu i když se identifikátor a klíčové slovo liší pouze velikostí písmen. Avšak při zápisu klíčového slova se musí dodržet ve shodě se specifikací velikost znaků. V opačném případě může překladač hlásit upozornění (warning).

a) Příklad:
boolean je platné klíčové slovo.
Boolean a *BOOLEAN* jsou nepovolené zápisy klíčových slov (liší se velikostí znaků) a zároveň nepovolené identifikátory (kolidují s *boolean*).

b) Příklad:
Příklad poukazuje na způsob vyhodnocení tzv. *escape identifikátorů.*
typedef string a;
typedef string _a; // _a je vyhodnoceno jako by bylo jednoduše a, překladač hlásí chybu "duplicate definition of a".

2.4.3 Komentáře

Komentáře se v jazyce *IDL* zapisují stejně jako například v jazyce C++ nebo Java. Existuje možnost zakomentovat samostatný řádek nebo jeho část, stejně tak je možné zakomentovat několikařádkový úsek kódu. Jednořádkový komentář začíná dvojicí znaků slash (//) a jeho platnost je až do konce řádky. Víceřádkový komentář začíná dvojicí znaků slash a hvězdička (/*) a končí stejnou dvojicí ale v obráceném pořadí (*/).

Příklad:

```
// Jednořádkový komentář
/*  Víceřádkový
    komentář   */
```

2.4.4 Konstanty

Konstanty se deklarují podobně jako v jiných jazycích pomocí klíčového slova *const*.

2.4.5 Základní datové typy, typ string a wstring

Přehled základních datových typů jazyka IDL je uveden v tab. 7. V tabulce je také uvedeno jejich mapování do jazyka Java a C++.

Tab. 7: Přehled základních datových typů jazyka IDL

IDL	Java	C++
boolean	boolean	CORBA::Boolean
char	char	CORBA::Char
wchar	char	CORBA::WChar
octet	byte	CORBA::Octet
string	java.lang.String	viz. [8]
wstring	java.lang.String	viz. [8]
short	short	CORBA::Short
unsigned short	short	CORBA::UShort
long	int	CORBA::Long
unsigned long	int	CORBA::ULong
long long	long	CORBA::LongLong
unsigned long long	long	CORBA::ULongLong
float	float	CORBA::Float
double	double	CORBA::Double
fixed	java.math.BigDecimal	viz. [8]

2.4.6 Strukturované typy, speciální typy

Budou uvedena pouze základní pravidla použití pro základní strukturované a speciální typy, protože popis všech typů by v textu zabral přespříliš místa. Popis deklarace strukturovaných typů je pomocí EBNF (Extended Backus-Naur Form) uveden ve specifikaci [1]. Popis všech speciálních typů je také uveden v [1].

Strukturované typy

2.4.6.1 Záznam (struct)

Záznamy jsou nehomogenní datové typy. Skládájí se z položek, které jsou určeny identifikátorem a typem. Typy položek se nemusí shodovat a typem položky mohou být dokonce i strukturované typy.

Speciální typy

2.4.6.2 Pole

Pole se deklaruje obdobně jako v jazyce C. Deklarace začíná typem prvků pole, následuje mezera, identifikátor pole a za ním uvnitř hranatých závorek délka pole. Délka pole je pevná, pole obecně nelze prodlužovat, pokud budeme chtít pole prodloužit měli bychom spíše použít IDL typ *sekvence*.

2.4.6.3 Sekvence (sequence)

Speciální typ *sekvence* představuje jednorozměrné pole s proměnnou délkou. Při deklaraci se použije klíčové slovo *sequence*. Jako volitelný parametr je možné definovat maximální délku neboli kapacitu pole. V tom případě se jedná o ohraničenou sekvenci (*bounded sequence*). Jestliže parametr vynecháme, bude se jednat o neohraničenou sekvenci (*unbounded sequence*). Je-li kapacita určena parametrem jako pevná (*bounded sequence*), překladač určí velikost potřebné paměti již během překladu. V opačném případě (*unbounded sequence*) se bude velikost potřebné paměti určovat až za běhu programu, podle aktuální délky *sekvence*.

2.4.6.4 Typ Any (Any)

Objektový typ *Any* má v *IDL* zvláštní význam. Je to kontejner (obalující třída) do kterého je možné ukládat (a následně z něj extrahovat) hodnotu jakéhokoli typu. Typ *Any* umožňuje současně hodnotu v jeho objektu zapsanou dynamicky identifikovat za běhu programu. Díky zapouzdření hodnot různých typů typem *Any* je možné získat množinu hodnot, které se navenek jeví se stejnými vlastnostmi. Díky tomu je můžeme následně například ukládat do sekvence či pole nebo s nimi jinak jednotně manipulovat.

2.4.7 Obory platnosti identifikátorů

Obor platnosti identifikátoru je v *IDL* od místa deklarace identifikátoru do konce tzv. *scope* ve kterém byla tato deklarace uvedena, tj. v tomto rozsahu identifikátor jednoznačně určuje jeho neúplně kvalifikované jméno. Chceme-li použít identifikátor mimo tento rozsah platnosti, musíme použít plně kvalifikovaného jména identifikátoru.

V *IDL* k tomu slouží operátor čtyřtečka (::). Plně kvalifikovaného jména identifikátoru využijeme i v některých případech zastínění deklarace nebo nejednoznačnosti (obvykle vzniklé při importu více knihoven obsahujících stejný identifikátor pomocí direktivy #include).

2.4.7.1 Jmenné prostory (module)

Klíčové slovo *module* slouží v jazyce *IDL* k vymezení jmenného prostoru. Jmenné prostory usnadňují správu identifikátorů ve zdrojovém kódu a zamezují kolizím v souvislosti s jejich používáním. Ekvivalenty klíčového slova *module* v Javě a v C++ jsou klíčová slova *package*, resp. *namespace*.

2.4.7.2 Rozhraní (interface)

Deklarace rozhraní v IDL (klíčové slovo *interface*) odpovídá deklaraci abstraktní třídy v C++ resp. rozhraní v Javě. Co všechno můžeme v rámci klíčového slova *interface* zapsat viz. [1].

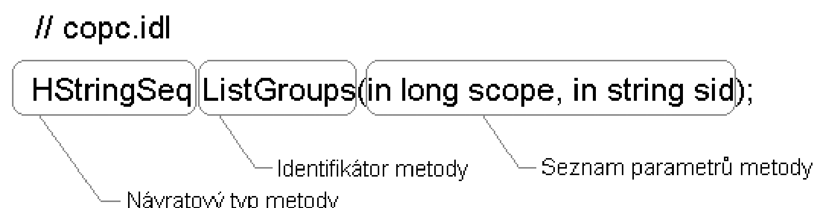
2.4.8 Deklarace nového typu

K definici nového pojmenovaného typu slouží v IDL klíčové slovo *typedef*. Deklaraci nového typu si ukážeme na příkladech.

```
// copc.idl
...
typedef long HRESULT;
...
typedef sequence<unsigned long> ULongSeq;
...
typedef struct HString {
  HRESULT hr;
  string str;
} str_hstring;
...
```

2.4.9 Deklarace operace - metody

Rozbor deklarace operace je uveden na příkladě na obr. 6.



Obr. 6: Deklarace operace v jazyce IDL

2.4.10 Parametry funkcí, návratový typ funkce

V hlavičce metody můžeme jak bylo popsáno v předchozím odstavci deklarovat také parametry. Parametry metod se deklarují podobně jako např. v Javě. Deklarace začíná specifikátorem *in*, *inout* nebo *out*, dále se uvede typ parametru a jeho identifikátor. Jednotlivé deklarace parametrů se oddělují čárkou. Pokud metoda nepřijímá žádné parametry, je potřeba uvést v seznamu parametrů klíčové slovo *void*. Příklad viz. odst. *2.4.9 Deklarace operace - metody*.

2.4.11 IDL Rozhraní CORBA-OPC můstku

Pro příklad použití jazyka IDL je možné se podívat na deklaraci IDL rozhraní CORBA-OPC můstku v souboru *copc.idl* v adresáři *soft/src/copc/* na přiloženém CD-ROM, resp. popis typů a metod tam uvedených je uveden také v odst. 3.1.3.2.

2.5 CORBA - Používaná hesla

IDL Stub ... objekt klienta, delegující žádosti na server. *IDL Stub* poskytuje klientovi identické rozhraní služeb (metod) serveru. *IDL Stub* se uplatňuje při statické komunikaci klient-server. *IDL Stub* může také používat rozhraní *DII*, viz. heslo Dy*namic Invocation Interface*. Viz. také odst. *2.3.4.1 IDL Stub*.

IDL Skeleton ... objekt serveru, implementující statický, tzv. 'dispatching' (předávací) mechanismus pro klientské požadavky. Jiný způsob předávání je dynamický, využívající tzv. *dynamický skeleton*. *IDL Skeleton* se také uplatňuje při statické komunikaci klient-server. Viz. také odst. *2.3.4.2 IDL Skeleton*.

Dynamický skeleton ... obdoba *IDL Skeletonu* pro dynamickou komunikaci klient-server, využívá rozhraní *DSI*, viz. heslo *Dynamic Skeleton Interface*.

Dynamic Invocation Interface (DII) ... část aplikačního rozhraní *corby*, umožňující dynamické sestavení požadavku při komunikaci (rozhraní služeb serveru je známo až v době běhu aplikace). Některé metody rozhraní *DII* jsou uvedeny v odstavci *2.3.1 ORB*. Rozhraní *DII* obvykle využívá klient.

Dynamic Skeleton Interface ... část aplikačního rozhraní *corby*, umožňující dynamický 'dispatching' mechanismus, viz. heslo *Dynamický skeleton* a viz. odst. *2.3.5.2 Rozhraní Dynamic Skeleton Interface*.

ORB (Object Request Broker) ... základní komponenta v *corbě*. *ORB* zajišťuje všechny základní funkce pro komunikaci mezi klientem a serverem. *ORB* klienta a serveru spolu obvykle komunikují pomocí protokolu *IIOP*, viz. heslo *IIOP* a viz. také odst. *2.3.1 ORB*.

IIOP (Internet Inter ORB Protocol) ... protokol, odvozený od obecnějšího vzoru *GIOP* (*General Inter ORB Protocol*), využívající TCP/IP konekce. Viz. také [1].

Objekt (Objektova implementace) ... objekt (obvykle ve smyslu třídy) implementující služby (metody) serveru.

Objektový adaptér ... objektové rozhraní mezi *orbem* a *objektovou implementací*. *ORB* obvykle nemanipuluje s objektem přímo, ale prostřednictvím *objektového adaptéru*, viz. odst. *2.3.2 Objektový adaptér*.

POA (Portable Object Adapter) ... konvenční *objektový adaptér* v *corbě*, uvedený ve specifikaci [1]. Nahrazuje dřívější konvenční objektový adaptér *BOA* (*Basic Object Adapter*).

Objektová reference ... základní způsob vzdáleného odkazování na objekt. *Objektová reference* obsahuje informace o vzdáleném objektu (např. lokalizační) a umožňuje tak klientovi sestavit požadavek a směrovat požadavek v síti. Viz. také *2.2.1 Objektová reference*.

Objektová služba ... základní koncept *corby*. *Objektové služby* mohou být integrální součástí CORBA aplikace, anebo mohou být řešeny jako samostatné aplikace, ke kterým přistupuje klient či server. Zpravidla implementují pomocné a užitečné funkce. Příkladem jsou *jmenná služba* a služba *Interface Repository*.

Servant ... synonymum pro instanci objektové implementace (objektu).

Servant Manager ... *Servant managera* používá *objektový adaptér* v případech, kdy není schopen běžným způsobem lokalizovat *servanta* na serveru. Programátor může implementovat vlastního *servant managera* a umožnit tak *objektovému adaptéru* lokalizovat *servanta* specifickým způsobem. Existují dva typy *servant managerů*, a to *PortableServer::ServantActivator* a *PortableServer::ServantLocator*, viz. také *2.3.2.4 Vyhledání Servanta*.

POA Manager ... viz. *2.3.2.11 POA manager*.

Jmenná služba ... *objektová služba* v *corbě*, která slouží pro podporu evidence a lokalizace *objektových implementací* v síti, viz. odst. *2.3.3 Jmenná služba*.

Interface Repository (IR) ... objektová služba v *corbě*. Slouží pro podporu evidence informací o *objektech* (typu jaké rozhraní a metody implementuje, jaké jsou typy parametrů metod apod.), je využitelná pro dynamickou identifikaci *objektů* a může být využita také pro typovou kontrolu. Viz. také odst. *2.3.5.3 Služba Interface Repository*.

OMG IDL (OMG Interface Definition Language) ... deklarativní jazyk, specifikovaný skupinou OMG ([11]). Používá se pro jazykově nezávislé deklarování rozhraní objektů. Jazyk *OMG IDL* se snaží zachytit deklarativní rysy jiných jazyků (Java, C++, Smalltalk, ...) a postihnout je ve své syntaxi a tím je zobecnit a sjednotit. Viz. odst. *2.4 Jazyk OMG IDL*.

3 Praktická část - Implementace PPVV

3.1 CORBA-OPC můstek (C-OPC)

První aplikací, navrženou v rámci diplomové práce, je softwarový můstek pro účely propojení technologií CORBA a OPC. Návrh této aplikace byl základním krokem pro to, aby byl CORBA klientům umožněn přístup k datům uložených na OPC serverech.V mé práci zastává funkci CORBA klienta vizualizační applet (viz. odst. 3.3), který za účelem získání technologických dat kontaktuje navržený CORBA-OPC můstek a využívá můstek jako CORBA server. Tento můstek je zároveň také OPC klientem, který za účelem sběru technologických dat kontaktuje OPC server.

3.1.1 Základní charakteristika

C-OPC funguje jako konzolová aplikace bez grafického uživatelského rozhraní. Po spuštění umožňuje C-OPC nastavit pracovní parametry. Poté přechází do pracovního stavu, kdy zpracovává požadavky klientů.

3.1.2 Použité technologie - shrnutí

CORBA-OPC můstek jsem implementoval v jazyce C++ ve vývojovém prostředí *Borland C++ Builder 5.0*. Pro komunikaci applet-můstek je použita CORBA implementace, známá jako *Visibroker for C++ 4.0*, jenž je součástí balíku *C++ Builderu*. Pro komunikaci můstek-OPC server je použita technologie OPC (COM).

3.1.2.1 COM a OPC

Protože na katedře již byla řešena diplomová práce zabývající se návrhem OPC klienta, a protože tato práce obsahuje souhrn všech základních poznatků potřebných pro pochopení jeho návrhu, v mé práci se již nebudu dalekosáhle rozepisovat ani o technologii OPC, ani o technologii COM, na které je OPC vystavěno. Zmíním pouze některé základy pro zopakování, pro podrobnosti zájemce odkazuji na informace v [6] a [12].

Základní podmínkou pro to, abychom mohli na daném systému vytvářet instance COM objektů, je, aby měl implementovanou podporu technologie COM. Pokud je tato podmínka splněna a na systému máme instalovány nějaké COM objekty, můžeme vytvářet jejich instance.

COM objekt obvykle zastupuje roli serveru a zpřístupňuje své rozhraní klientu, který jeho rozhraní volá. Instance COM objektu tedy vytváří klient. Podle toho, zda klient a server sdílejí stejný proces nebo ne, hovoříme o *in-process* serveru, resp. *out-of-process* serveru. Klient, který pracuje s *in-process* serverem, vytváří jeho instanci a volá metody jeho rozhraní přímo.

Klient, který pracuje s *out-of-process* serverem, nevytváří přímo jeho instanci, ale jeho *proxy* objekt, a nevolá metody jeho rozhraní přímo, ale pomocí tohoto *proxy* objektu. *Proxy* objekt provádí tzv. *marshalling* zpráv a využívá pro komunikaci se vzdáleným objektem volání *RPC* (*Remote Procedure Call*). V každém případě klient vytváří instanci COM serveru nebo jeho *proxy* stejně a transparentně také přistupuje k jeho rozhraní. Proto z hlediska klienta není nutné rozlišovat mezi *in-process* nebo *out-of-process* serverem.

Poznámka:
OPC klient a OPC server jsou jednoduše řečeno COM klient a COM server, rozdíl spočívá v tom, že OPC vytváří více specifický model použitelný pro průmyslovou automatizaci. Definuje především standardní, jednotné rozhraní pro OPC servery a klienty, a standardizuje další pravidla a podrobnosti pro návrh OPC aplikací. Některé funkce a typy OPC rozhraní budou uvedeny v tabulkách 6 a 8 v odstavci *3.1.3.2 Serverové rozhraní můstku - IDL rozhraní*.

Vlákno COM klienta (OPC klienta) musí nejprve vstoupit do tzv. COM apartmentu, než může volat metody COM, vytvářet COM objekty a následně volat jejich rozhraní. Aby vlákno klienta získalo COM apartment, volá metodu *CoInitialize()*, resp. *CoInitializeEx()*. Následně, po ukončení práce s COM objektem, vlákno volá metodu *CoUninitialize()* a apartment opouští. Poté co vlákno získá apartment, může vytvořit COM objekt. Po vytvoření instance COM objektu je klientovi vrácen pointer na jeho rozhraní a klient pomocí rozhraní pracuje s objektem a volá jeho metody.

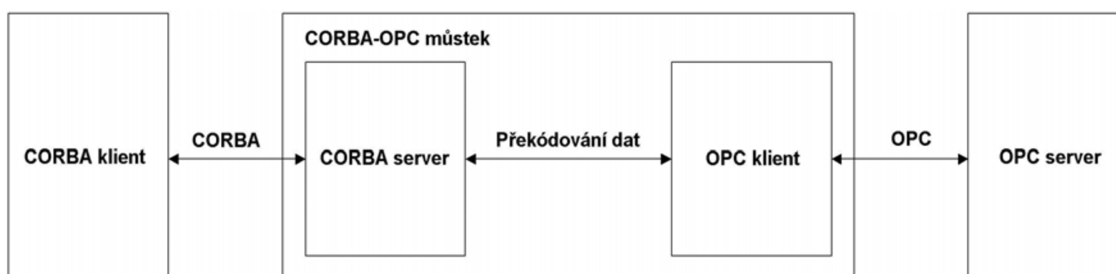
3.1.3 Podrobnosti implementace

3.1.3.1 CORBA a COM spolupráce

Při návrhu takového můstku vyvstávají určité otázky, zejména týkající se vzájemné slučitelnosti CORBY a OPC. Proto bylo nutné se nejprve pokusit identifikovat možné problémy, které by znemožnily realizaci takového můstku.

Základním problémem bylo, jak přenést data z OPC serveru až ke CORBA klientovi. OPC pracuje se speciálními datovými typy a strukturami definovanými na základě typů COM. Přenést data v tomto tvaru není možné přímo. V praxi prozatím neexistuje žádný standard, který by deklaroval mapování datových typů mezi technologiemi OPC a CORBA.

Proto bylo nutné na úrovni můstku provést rozklad struktur a objektových typů používaných v OPC na elementární typy. Následně jsem pak deklaroval nové typy v jazyce OMG IDL které obalily původní OPC typy. Můstek tedy provádí jako jednu ze svých základních činností překódování typů z OPC na typy technologie CORBA, viz. obr. 7, tab. 8 a 9.



Obr. 7: Činnost CORBA-OPC můstku

3.1.3.2 Serverové rozhraní můstku - IDL rozhraní

Můstek, jako CORBA server, poskytuje klientům rozhraní pro přístup k OPC datům. Toto rozhraní můstku jsem deklaroval v jazyce OMG IDL (viz. odst. *2.4 Jazyk OMG IDL*), tedy nezávisle na použitém implementačním jazyce. IDL rozhraní můstku je ve zdrojové formě dostupné na přiloženém CD-ROM v souboru *copc.idl* v adresáři

soft/src/copc/. Následně bylo možné zvolit implementační jazyk můstku, zvolil jsem C++, teoreticky by však bylo možné zvolit i jiný jazyk s podporpou COM a CORBY. Další postup spočíval v překladu IDL rozhraní můstku do jazyka C++ (s použitím překladače dostupného v *C++ Builderu idl2cpp.exe*) jehož výstupem byla základní třída (IDL Skeleton) pro implementaci můstku. Do této třídy bylo potřeba doimplementovat metody pro přístup k OPC serveru a překódování datových typů OPC a CORBA pro jejich obousměrný přenos.

Implementoval jsem pouze některé metody OPC rozhraní *IOPCServer*, *IOPCItemMgt*, funkci synchronního čtení *IOPCSyncIO::Read()* a metody rozhraní *IOPCItemAttributes* a *IUnknown*, tedy zejména ty, které využiji pro vizualizaci. Neimplementoval jsem například funkce asynchronního čtení a zápisu, metodu synchronního zápisu a některé funkce informačního charakteru a některé nepovinné OPC rozhraní.

V následující tabulce 8. a 10. jsou typy a metody deklarované v OPC a jejich odpovídající ekvivalenty, které jsem deklaroval v jazyce OMG IDL. Navíc bylo nutné přidat ještě další metody, viz. tab. 11.

Poznámka:
Jména ekvivalentních typů, které jsou uvedeny ve výše zmíněných tabulkách jsou většinou konstruována jako [H]C<opc type name>. Typy začínající prefixem H určují, že typ je návratový.

Tab. 8: OPC typy a jejich IDL ekvivalenty v CORBA-OPC můstku

OPC typy	**Ekvivalenty deklarované v OMG IDL**
typedef unsigned long **LCID**;	typedef unsigned long **LCID**;
typedef long **HRESULT**;	typedef long **HRESULT**;
typedef struct { [string] LPWSTR szAccessPath; [string] LPWSTR szItemID; BOOL bActive; OPCHANDLE hClient; OPCHANDLE hServer; DWORD dwAccessRights; DWORD dwBlobSize; [size_is(dwBlobSize)] BYTE * pBlob; VARTYPE vtRequestedDataType; VARTYPE vtCanonicalDataType; OPCEUTYPE dwEUType; VARIANT vEUInfo; } **OPCITEMATTRIBUTES**;	typedef struct **COPCITEMATTRIB** { string accessPath; string itemID; boolean active; unsigned long clientHandle; unsigned long serverHandle; unsigned long accessRights; TBLOB blob; long requestedDataType; long canonicalDataType; short euType; any euInfo; } str_copcitemattrib;

<table>
<tr>
<td>typedef struct {
FILETIME ftStartTime;
FILETIME ftCurrentTime;
FILETIME ftLastUpdateTime;
OPCSERVERSTATE dwServerState;
DWORD dwGroupCount;
DWORD dwBandWidth;
WORD wMajorVersion;
WORD wMinorVersion;
WORD wBuildNumber;
WORD wReserved;
[string] LPWSTR szVendorInfo;
} OPCSERVERSTATUS;</td>
<td>typedef struct HCOPCSERVERSTATUS {
HRESULT hr;
unsigned long startTimeHi;
unsigned long startTimeLo;
unsigned long currentTimeHi;
unsigned long currentTimeLo;
unsigned long lastUpdateTimeHi;
unsigned long lastUpdateTimeLo;
unsigned long serverState;
unsigned long groupCount;
unsigned long bandWidth;
unsigned short majorVersion;
unsigned short minorVersion;
unsigned short buildNumber;
unsigned short reserved;
string vendorInfo;
} str_hcopcserverstatus;</td>
</tr>
<tr>
<td>typedef struct {
[string] LPWSTR szAccessPath;
[string] LPWSTR szItemID;
BOOL bActive;
OPCHANDLE hClient;
DWORD dwBlobSize;
[size_is(dwBlobSize)] BYTE * pBlob;
VARTYPE vtRequestedDataType;
WORD wReserved;
} OPCITEMDEF;</td>
<td>typedef struct COPCITEMDEF {
string accessPath;
string itemID;
boolean active;
unsigned long clientHandle;
TBLOB blob;
long dataType;
unsigned short reserved;
} str_copcitemdef;</td>
</tr>
<tr>
<td>typedef struct {
OPCHANDLE hClient;
FILETIME ftTimeStamp;
WORD wQuality;
WORD wReserved;
VARIANT vDataValue;
} OPCITEMSTATE;</td>
<td>typedef struct COPCITEMSTATE {
unsigned long clientHandle;
unsigned long timeStampHi;
unsigned long timeStampLo;
unsigned short quality;
unsigned short reserved;
any dataValue;
} str_copcitemstate;</td>
</tr>
<tr>
<td>typedef struct {
OPCHANDLE hServer;
VARTYPE vtCanonicalDataType;
WORD wReserved;
DWORD dwAccessRights;
DWORD dwBlobSize;
[size_is(dwBlobSize)] BYTE * pBlob;
} OPCITEMRESULT;</td>
<td>typedef struct COPCITEMRESULT {
unsigned long handle;
long dataType;
unsigned short reserved;
unsigned long accessRights;
TBLOB blob;
} str_copcitemresult;</td>
</tr>
</table>

Tab. 9: Pomocné typy v CORBA-OPC můstku

Pomocné typy deklarované v OMG IDL
typedef sequence<HRESULT> **HRESULTSeq**;
typedef sequence<unsigned long> **ULongSeq**;
typedef sequence<octet> **TBLOB**;
typedef sequence<string> **StringSeq**;
typedef sequence<COPCITEMATTRIB> **COPCITEMATTRIBSeq**;
typedef sequence<COPCITEMDEF> **COPCITEMDEFSeq**;
typedef sequence<COPCITEMRESULT> **COPCITEMRESULTSeq**;
typedef sequence<COPCITEMSTATE> **COPCITEMSTATESeq**;
typedef struct **HString** { HRESULT hr; string str; } str_hstring;
typedef struct **HULong** { HRESULT hr; unsigned long val; } str_hulong;
typedef struct **HStringSeq** { HRESULT hr; StringSeq seq; } str_hstringseq;
typedef struct **HResAddGroup** { HRESULT hr; unsigned long serverGroup; unsigned long revisedRate; unsigned long rid; } str_hresaddgroup;
typedef struct **HResRead** { HRESULT hr; COPCITEMSTATESeq itemsDataSeq; HRESULTSeq hSeq; } str_hresread;
typedef struct **HResAddItems** { HRESULT hr; COPCITEMRESULTSeq rSeq; HRESULTSeq hSeq; } str_hresadditems;
typedef struct **HCOPCITEMATTRIBSeq** { HRESULT hr; COPCITEMATTRIBSeq seq; } str_hcopcitemattribseq;

Tab. 10: OPC metody a jejich IDL ekvivalenty v CORBA-OPC můstku

HRESULT **GetErrorString**(/* [in] */ HRESULT dwError, /* [in] */ LCID dwLocale, /* [string][out] */ LPWSTR __RPC_FAR *ppString);	HString **IOPCServer_GetErrorString**(in HRESULT hr, in LCID lcid, in string sid);
HRESULT **GetStatus**(/* [out] */ OPCSERVERSTATUS __RPC_FAR *__RPC_FAR *ppServerStatus);	HCOPCSERVERSTATUS **IOPCServer_GetStatus**(in string sid);
HRESULT /* IOPCServer */ **AddGroup**(/* [string][in] */ LPCWSTR szName, /* [in] */ BOOL bActive, /* [in] */ DWORD dwRequestedUpdateRate, /* [in] */ OPCHANDLE hClientGroup, /* [in][unique] */ LONG __RPC_FAR *pTimeBias, /* [in][unique] */ FLOAT __RPC_FAR *pPercentDeadband, /* [in] */ DWORD dwLCID, /* [out] */ OPCHANDLE __RPC_FAR *phServerGroup, /* [out] */ DWORD __RPC_FAR *pRevisedUpdateRate, /* [in] */ REFIID riid, /* [iid_is][out] */ LPUNKNOWN __RPC_FAR *ppUnk);	HResAddGroup **IOPCServer_AddGroup**(in string groupName, in boolean active, in unsigned long rate, in unsigned long clientGroup, in long timeBias, in float deadBand, in LCID lcid, in string riid, in string sid);
HRESULT /* IOPCServer */ **GetGroupByName**(/* [string][in] */ LPCWSTR szName, /* [in] */ REFIID riid, /* [iid_is][out] */ LPUNKNOWN __RPC_FAR *ppUnk);	HULong **IOPCServer_GetGroupByName**(in string name, in string riid, in string sid);

HRESULT /* IOPCServer */ **CreateGroupEnumerator**(/* [in] */ OPCENUMSCOPE dwScope, /* [in] */ REFIID riid, /* [iid_is][out] */ LPUNKNOWN __RPC_FAR *ppUnk);	HULong **IOPCServer_CreateGroupEnumerator**(in long scope, in string riid, in string sid);
HRESULT /* IOPCItemMgt */ **AddItems**(/* [in] */ DWORD dwCount, /* [size_is][in] */ OPCITEMDEF __RPC_FAR *pItemArray, /* [size_is][size_is][out] */ OPCITEMRESULT __RPC_FAR *__RPC_FAR *ppAddResults, /* [size_is][size_is][out] */ HRESULT __RPC_FAR *__RPC_FAR *ppErrors);	HResAddItems **IOPCItemMgt_AddItems**(in unsigned long rid, in COPCITEMDEFSeq itemSeq, in string sid);
HRESULT /* IOPCItemMgt */ **CreateEnumerator**(/* [in] */ REFIID riid, /* [iid_is][out] */ LPUNKNOWN __RPC_FAR *ppUnk);	HULong **IOPCItemMgt_CreateEnumerator**(in unsigned long rid, in string riid, in string sid);
HRESULT /* IOPCSyncIO */ **Read**(/* [in] */ OPCDATASOURCE dwSource, /* [in] */ DWORD dwCount, /* [size_is][in] */ OPCHANDLE __RPC_FAR *phServer, /* [size_is][size_is][out] */ OPCITEMSTATE __RPC_FAR *__RPC_FAR *ppItemValues, /* [size_is][size_is][out] */ HRESULT __RPC_FAR *__RPC_FAR *ppErrors);	HResRead **IOPCSyncIO_Read**(in unsigned long rid, in long source, in ULongSeq itemHandles, in string sid);
/* IUnknown */ HRESULT **QueryInterface**(REFIID iid, void ** ppvObject);	HULong **IUnknown_QueryInterface**(in unsigned long rid, in string riid, in string sid);

/* IEnumOPCItemAttributes */ HRESULT **Next**(/* [in] */ ULONG celt, /* [size_is][size_is][out] */ OPCITEMATTRIBUTES __RPC_FAR *__RPC_FAR *ppItemArray, /* [out] */ ULONG __RPC_FAR *pceltFetched);	HCOPCITEMATTRIBSeq **IEnumOPCItemAttributes_Next**(in unsigned long rid, in unsigned long celt, in string sid);
/* IEnumOPCItemAttributes */ HRESULT **Skip**(/* [in] */ ULONG celt);	HRESULT **IEnumOPCItemAttributes_Skip**(in unsigned long rid, in unsigned long celt, in string sid);
/* IEnumOPCItemAttributes */ HRESULT **Reset**(void);	HRESULT **IEnumOPCItemAttributes_Reset**(in unsigned long rid, in string sid);

Tab. 11: Pomocné metody v CORBA-OPC můstku

Pomocné metody deklarované v OMG IDL
HStringSeq **ListGroups**(in long scope, in string sid);
HRESULT **Connect**(in string sid);
HRESULT **Disconnect**(in string sid);
HRESULT **OpcActive**(in string sid);

3.1.3.3 Práce s pamětí pro přenášená data

Paměť pro výsledky volání OPC metod alokuje OPC server. V souladu se zvyklostmi a zásadami modelu COM, OPC klient by měl vždy uvolnit paměť, kterou alokuje OPC server pro návratové objekty OPC metod. OPC klient by měl po použití uvolnit paměť pro hodnoty všech parametrů, označených v deklaracích OPC metod jako *out*, resp. *inout*. V případě komunikace je to zvlášť kritický požadavek, protože při déle trvající komunikaci by v takovém připadě mohlo dojít k vyčerpání paměti a zhroucení komunikace.

3.1.3.4 Vláknový model CORBA-OPC můstku

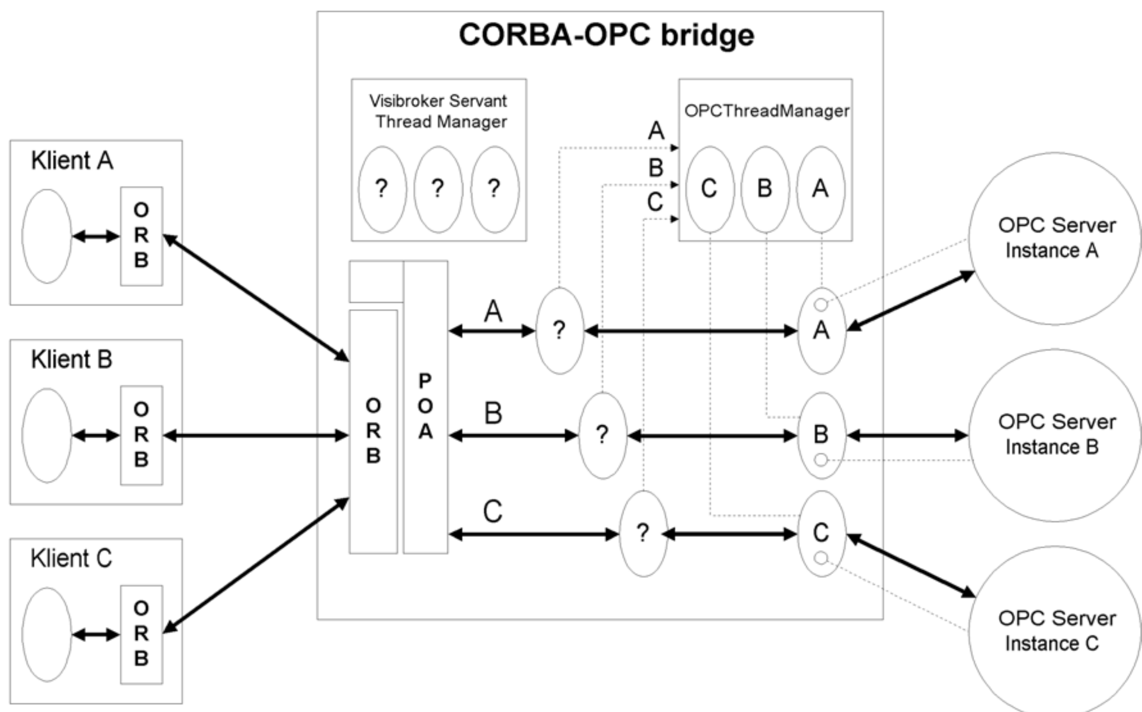
K můstku se může v jeden okamžik připojit více klientů najednou, proto bylo potřeba se při návrhu zabývat také vlákny pro zpracování klientských požadavků.

Vlákno OPC klienta udržuje pointery na rozhraní OPC serveru. Zpravidla OPC klient pracuje v rámci všech operací s OPC serverem v jednom a tom samém vlákně. V opačném případě by musel volat metody pro získání COM apartmentu a vytvářet instanci OPC serveru pro každé nové vlákno. Jak jsem zjistil, tyto operace jsou poměrně časově náročné. Proto je žádoucí aby CORBA-OPC můstek vytvářel pro každého

CORBA klienta pouze jedno vlákno OPC klienta a to samé vlákno pro něj udržoval pro různé požadavky.

Tady jsem přišel na jisté obtíže v souvislosti se správou vláken v CORBA implementaci *Visibroker*, použité pro návrh C-OPC. *Visibroker* umožňuje vytvořit pro každého klienta vlákno pro zpracování požadavků (viz. dokumentace [4], Chapter 8: Managing threads and connections, Thread-per-session policy). Podle dokumentace by při správném nastavení mělo být vždy jedno a to samé vlákno použito pro zpracování všech požadavků od klienta. Bohužel se mi nakonec nepodařilo zařídit, aby tato požadovaná situace nastala. Při testování jsem vždy nakonec zjistil, že vlákna mohou být i různá. Po dlouhém úsilí jsem nakonec přistoupil k náhradnímu řešení.

C-OPC vytváří při každém spuštění jednoho servanta, každý požadavek však zpracuje v samostatném vlákně. Jelikož se mi nepodařilo dosáhnout činnosti vláken podle výše zmíněné situace, definoval jsem pro vytváření a správu vláken OPC klientů třídu OPCThreadManager. C-OPC můstek vytváří pouze jednu instanci třídy OPCThreadManager a tu používá pro správu vláken a získání správného vlákna pro požadavek. Požadavky jsou označeny příznakem *sessionID*, který je pro každého klienta unikátní a klient udržuje tento příznak platný po celou dobu komunikace se serverem. Servant tak může snadno rozpoznat od kterého klienta požadavek přišel a vybere jemu odpovídající vlákno a s pomocí něj předá požadavek na OPC server. Tato situace je demosntrována na obr. 8. Vlákno servanta tedy zachycuje požadavky od CORBA klienta a předává je vláknu OPC klienta. Za tímto účelem spolu vlákna musí nějakým způsobem komunikovat.



Obr. 8: Vláknový model CORBA-OPC můstku

3.1.3.4.1 Komunikace vlákna servanta a OPC klienta

Vlákna spolu komunikují prostřednictvím binárních semaforů 'A' a 'B'. Způsob komunikace mezi vlákny s použitím binárního semaforu je možné si představit následovně. Každý semafor může nabývat jedné z hodnot 'červená' a 'zelená'. Vlákno

OPC klienta čeká na 'zelenou' semaforu 'A', kterou nastavuje vlákno servanta, poté co předá požadavek. Nyní, když má vlákno OPC klienta 'zelenou', zpracovává požadavek a nastaví semafor 'A' na 'červenou'. Vlákno servanta naopak čeká na 'zelenou' semaforu 'B', kterou nastavuje vlákno OPC klienta, poté co zpracuje požadavek a předá odpověď. Jakmile přijde 'zelená', vlákno servanta nastaví semafor 'B' na 'červenou' a vyzvedne výsledky. Jako semafory 'A' a 'B' jsem použil objekty typu TEvent z knihovny *C++ Builderu*. Požadavky od jednoho klienta jsou zpracovávány jeden po druhém.

3.1.3.4.2 Alokace vláken a timeout pro klienty

Administrátor serveru by měl mít možnost určit, zda je potřeba nastavit limit pro počet vláken zpracovávajících požadavky, např. z důvodu optimálního zatížení, nebo i z jiných důvodů. Když pak dosáhne počet vláken na serveru určitého počtu, server další vlákno nevytvoří a klient musí čekat, až jiný klient ukončí se serverem spojení.

Když ale od nějakého klienta nepřichází po delší dobu na server žádné požadavky, server by měl mít právo ukončit s klientem komunikaci, aby dal možnost dalšímu klientovi, který je nucen na uvolnění OPC vlákna čekat. Pro vylepšení funkce jsem do C-OPC začlenil funkce timeoutu a možnost nastavit maximální počet alokovaných vláken pro klienty.

3.1.4 Instalace a spouštění

Instalace spočívá v překopírování obsahu složky *soft/bin/copc* přiloženého CD-ROM média do požadované složky.

3.1.5 Popis konzoly, ovládání aplikace

Před tím, než spustíme server, je potřeba spustit *jmennou službu*. *Jmenná služba* může běžet na stejném počítači jako C-OPC, anebo i na jiném počítači v síti. Výchozí číslo portu je 900 anebo můžeme určit jiný port. Jméno hostitelského počítače a port jmenné služby pak určíme v C-OPC a v návrhovém prostředí, viz. tento odstavec a odst. *3.2.3.1.3 Uložení návrhu, export pro www*. Pro spuštění jmenné služby můžeme vytvořit jednoduchý spouštěcí skript pro příkazový řádek. Obsah skriptu by mohl vypadat následovně:

```
set TNAMEPATH=c:\j2sdk1.4.1_02\bin
%TNAMEPATH%\tnameserv -ORBInitialPort 900
```

Po spuštění jmenné služby se zobrazí okno příkazového řádku s textem, jak je ukázáno na obr. 9.

```
C:\j2sdk1.4.1_02\bin\tnameserv.exe
Initial Naming Context:
```

```
IOR:000000000000002b49444c3a6f6d672e6f72672f436f734e616d696e672f4e616d696e67436f
6e746578744578743a312e300000000000000001000000000000008400010200000000 0f3136392e3235
342e32352e3132390000038400000000035afabcb0000000020ceee18a100000001000000000000
00010000000d544e616d655365727669636500000000000000040000000000a00000000000000010000
000100000002000000000000000100010020501000100010020000101090000000100010100
```


```
TransientNameServer: setting port for initial object references to: 900
Ready.
```

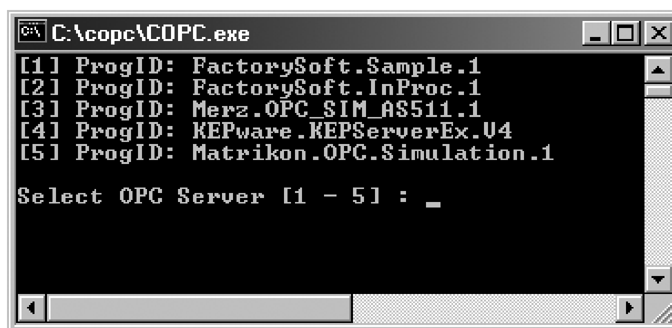
Obr. 9: Příkazový řádek po startu jmenné služby

Nyní můžeme spustit aplikaci C-OPC můstku (*copc.exe*). Zobrazí se rovněž okno konzole, viz. obr. 10.

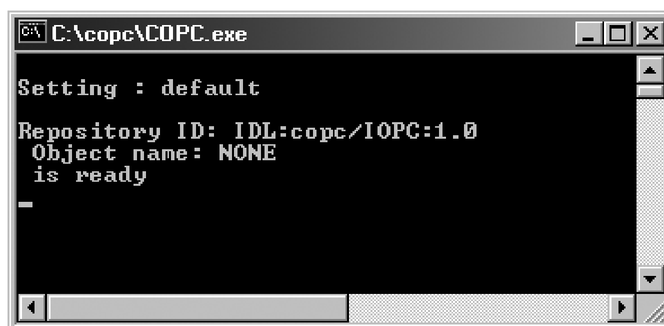
```
C:\copc\COPC.exe

      ****************************************
      *      CORBA to OPC bridge v. 1.0      *
      ****************************************

+-----------------------------------------------+
|             Default COPC settings:            |
+-----------------------+-----------------------+
|     Servant name:     |      OPCServant       |
+-----------------------+-----------------------+
|       POA name:       |    opcservant_poa     |
+-----------------------+-----------------------+
|       NS host:        |       localhost       |
+-----------------------+-----------------------+
|       NS port:        |          900          |
+-----------------------+-----------------------+
|     Max. threads:     |          20           |
+-----------------------+-----------------------+
|    Thread timeout:    |       20000 ms        |
+-----------------------+-----------------------+

Change default COPC settings? [y/n]: _
```

Obr. 10: Okno konzole po spuštění aplikace C-OPC můstku

V konzoli nyní vidíme výchozí nastavení pro C-OPC. Pokud tyto hodnoty nechceme měnit, potom na otázku pro změnu nastavení odpovíme záporně. Při kladné odpovědi se bude konzole postupně dotazovat na jednotlivé hodnoty. Poté co všechny otázky zodpovíme, C-OPC vylistuje seznam na počítači dostupných OPC serverů, podobně jako na obr. 11. Po výběru jednoho z nich by se měla zobrazit obrazovka s textovým výpisem jak ukazuje obr. 12. V tomto okamžiku C-OPC přešel do pracovního stavu a bude čekat na příchozí požadavky. Ukončit činnost C-OPC můstku můžeme po stisku klávesy 's' nebo 'S'.



Obr. 11: Výběr OPC serveru



Obr. 12: Okno konzole C-OPC po přechodu do pracovního stavu

3.1.6 Testování C-OPC můstku

Ve své práci se nezabývám návrhem OPC serveru. Činnost můstku jsem otestoval s pomocí testovací verze OPC serveru *KEPserver* společnosti *KEPware* a také s pomocí OPC serveru *WAGO OPC Server Modbus TCP*, dostupného pro PLC WAGO v laboratoři katedry. V podstatě by bylo možné použít jakýkoli dostupný OPC Server jak plyne ze samotné podstaty OPC. C-OPC můstek pracuje s oběma verzemi OPC serveru.

3.1.7 CORBA

Popis corby byl již uveden v teoretické části v kapitole 2. Zde uvedu pouze použití corby z hlediska praktického, tak jak je použita v C-OPC. Jako CORBA implementaci jsem použil implementaci *Visibroker for C++ 4.0* společnosti *Inprise*. Podrobné technické informace o této implementaci je možné nalézt v dokumentaci [3] a [4].

C-OPC můstek využívá statického modelu komunikace klient-server a používá tedy IDL stub a IDL skeleton. IDL rozhraní pro stub a skeleton je deklarováno v souboru *copc.idl* na přiloženém CD-ROM a přehled metod rozhraní již byl uveden v odstavci *3.1.3.2 Serverové rozhraní můstku - IDL rozhraní*. Po spuštění, poté co jsme nastavili výchozí hodnoty pro C-OPC jako jméno hostitelského počítače jmenné služby atd., a poté co jsme vybrali požadovaný OPC server, C-OPC vykoná postupně tyto kroky:

1. vytvoří a inicializuje instanci *orbu*
2. vytvoří *objektový adaptér POA*, kterému specifikuje požadované *POA politiky*
3. s adaptérem asociuje *POA manager*, který řídí zpracování požadavků
4. vytvoří instanci objektu - *servanta*, použitého pro zpracování všech požadavků od klientů

5. *servanta* asociuje s *objektovým adaptérem*
6. získá *root kontext jmenné služby*
7. pro *servanta* získá platnou *objektovou referenci*
8. *objektovou referenci* zapíše do báze *jmenné služby* (následně mohou klienti na daném hostitelském počítači a portu jmenné služby získat tuto referenci a volat metody referencovaného objektu)
9. aktivuje *POA manager*
10. zpracovává příchozí požadavky

Pro *POA adaptér* jsou použity v corbě implicitní politiky:

Thread Policy: ORB_CTRL_MODEL
Lifespan Policy: TRANSIENT
Object Id Uniqueness Policy: UNIQUE_ID
Id Assignment Policy: SYSTEM_ID
Servant Retention Policy: RETAIN
Request Processing Policy: USE_ACTIVE_OBJECT_MAP_ONLY
Implicit Activation Policy: NO_IMPLICIT_ACTIVATION

3.1.8 Shrnutí

Ačkoli můstek je implementován v jazyce *C++*, díky corbě je dostupný i pro klienty používající jiného jazyka, např. jazyk Java. V tomto jazyce jsem implementoval vizualizační applet. Výhodou appletu je, že je přístupný i pomocí běžně používaných www prohlížečů. Další podrobnosti o implementaci C-OPC můstku je možné nalézt na přiloženém CD-ROM.

3.1.9 Implementace CORBA serveru a klienta v prostředí Borland C++ Builder 5.0

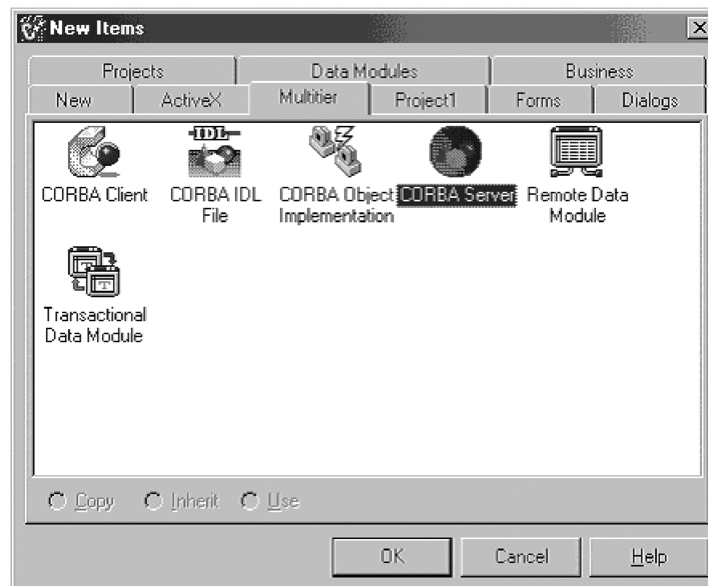
V tomto odstavci popíšu postup při vytváření CORBA aplikace typu server ve vývojovém prostředí *Borland C++ Builder 5.0*. Postup vytvoření klienta je téměř shodný, proto jej nebudu znovu uvádět, zmíním pouze odlišnosti.

Dříve než se zmíním o práci v prostředí, chtěl bych poznamenat, že *Visibroker* se instaluje samostatně. Pokud tedy chceme, aby byl hned po první instalaci součástí prostředí *Builderu*, nezapomeneme zatrhnout v instalátoru nabídku *'Nainstalovat Visibroker'*, která se objeví po skončení instalace *Builderu* a pokračovat v instalaci. Samozřejmě je možné jej doinstalovat později, avšak první postup je jednodušší. Po té co jsme nainstalovali *Visibroker*, budou v prostředí *Builderu* dostupné nové nástroje, vztahující se k návrhu aplikací v corbě.

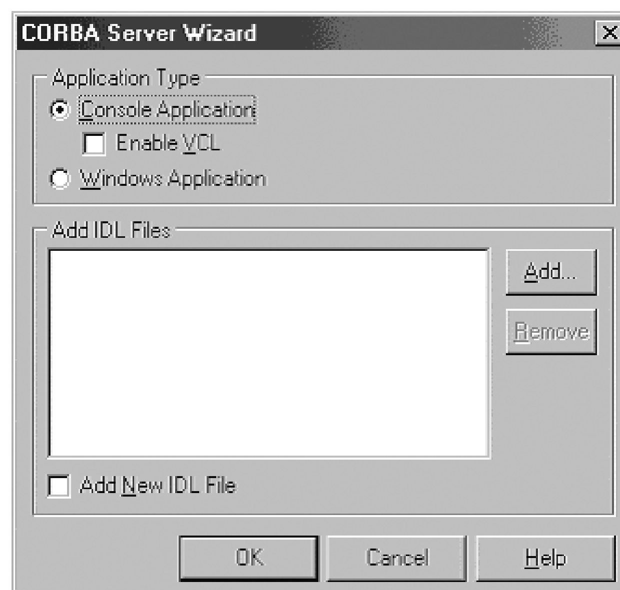
3.1.9.1 Vytvoření projektu pro aplikaci v corbě

Poté co běžným postupem spustíme prostředí *C++ Builderu* a rozhodneme se, že chceme vytvořit novou aplikaci v corbě, máme minimálně dvě možnosti jak pokračovat. Buď můžeme vybrat v menu nabídku 'File' a následně 'New...' a nebo v menu zvolit 'Project' a následně 'Add New Project...'. V obou případech se zobrazí stejné okénko 'New Items'. V tomto okně, viz. obr. 13, klikneme na záložku 'Multitier'. Zobrazí se okénko, které vidíme na obr. 13. Zde máme opět několik možností. Pokud chceme vytvořit samostatný OMG IDL soubor, který se dále uplatní při návrhu (např. pokud budeme používat IDL stub pro klienta, resp. IDL skeleton pro server) klikneme 2x na

ikonu označenou jako 'CORBA IDL file'. Tuto možnost je však možné přeskočit a soubor specifikovat jinak, viz. následující postup. Pokud budeme vytvářet klienta, klikneme 2x na ikonu 'CORBA Client'. Pokud budeme vytvářet server, klikneme 2x na ikonu 'CORBA Server'. Objeví se okno 'CORBA Server Wizard', ve kterém specifikujeme, zda budeme vytvářet konzolovou aplikaci anebo aplikaci windows. V tomto okně také specifikujeme OMG IDL soubor, který použijeme pro generování rozhraní IDL stubu a skeletonu, nebo můžeme nechat vytvořit nový, viz. obr. 14. Poté klikneme na 'OK'. Měli bychom vidět, že builder otevřel okno pro zdrojový kód serveru. Tento bude po překladu slinkován spolu s ostatními moduly do stejnojmenného 'exe' souboru. Můžeme vidět také záložku pro OMG IDL soubor, pokud jsme jej v předchozím kroku specifikovali. Nyní je dobré celý projekt uložit a určit tak složku ve které budou soubory projektu a zdrojové soubory.



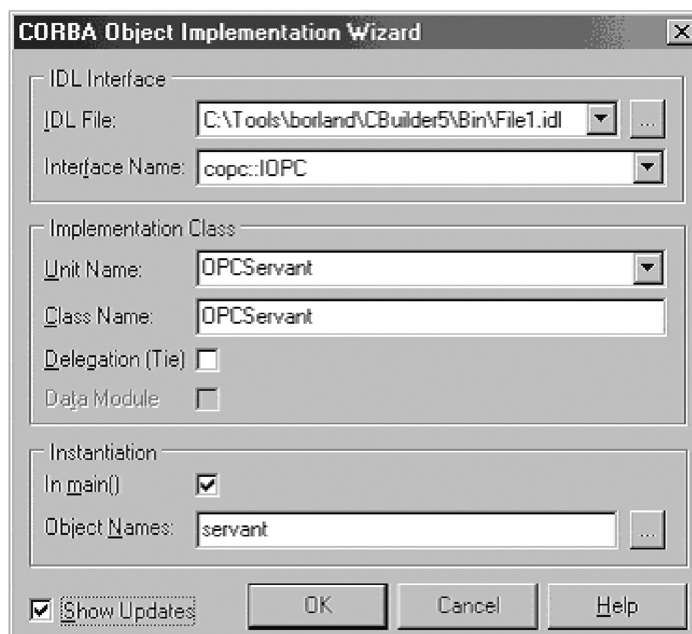
Obr. 13: Volby pro návrh CORBA aplikace v prostředí Borland C++ Builder



Obr. 14: Okno 'CORBA Server Wizard'

3.1.9.2 Vytvoření modulu pro objektovou implementaci

Pokud jsme specifikovali OMG IDL soubor, popisující rozhraní objektu implementujícího serverové služby, potřebujeme nyní vygenerovat základní kód pro IDL stub, resp. skeleton v jazyce C++. (V opačném případě, pokud jsme nespecifikovali OMG IDL soubor, jsme se rozhodli o dynamickou implementaci objektu a tedy nebudeme vytvářet IDL stub ani skeleton.) Použijeme proto následující postup. Přejdeme do okna 'New Items' a opět vybereme záložku 'Multitier'. Nyní však vybereme ikonu 'CORBA Object Implementation'. Objeví se okno 'CORBA Object Implementation Wizard', viz. obr. 15. V sekci IDL Interface určíme náš OMG IDL soubor a poté vybereme, které rozhraní má nový objekt implementovat. Dále v sekci 'Imlpementation Class' určíme jméno jednotky (Unit Name), která bude obsahovat třídu objektové implementace, jejíž název také uvedeme (Class Name). Nakonec v sekci 'Instantiation' můžeme určit jméno instance (Object Names) a zda má být instance vytvořena v metodě 'main'. Po té co jsme potvrdili (OK), bychom měli vidět, že *Builder* otevřel další okno pro zdrojový kód objektové implementace a v něm vidíme hlavičky (signatury) a prázdná těla přeložených metod, které formálně odpovídají deklaracím v souboru OMG IDL. Do nich budeme zapisovat implementační kód objektu. Při návrhu klienta objektovou implementaci nevytváříme.



Obr. 15: Okno 'CORBA Object Implementation Wizard'

3.1.9.3 Implementace

Nyní můžeme přistoupit k samotné implementaci. Detaily implementace se mohou lišit, proto uvedu pouze základní zásady. V metodě 'main' obvykle zapisujeme kód zajišťující vytvoření a inicializaci orbu, kód pro získání objektové reference na dostupné služby jako např. jmenné služby a objektové reference objektového adaptéru (POA) a specifikujeme politiky objektového adaptéru. Následně vytváříme instance objektových implementací (servanty), které registrujeme s požadovaným objektovým adaptérem. Dále je potřeba pro servanty vytvořit objektové reference, které můžeme např. zapsat do báze jmenné služby nebo do souboru ve formátu IOR a zajistíme jejich distribuci ke

klientovi. Nakonec po provedení všech potřebných základních akcí v 'main' uvedeme v činnost *orb* voláním jeho metody *run()*, která blokuje vlákno 'main' až do doby ukončení činnosti *orbu* voláním metody *shutdown()*. Druhou alternativou jak řídit předávání vlákna 'main' *orbu* je použití *polling smyčky*, příklad byl uveden v teoretické části v odst. *2.3.1.5 Metody pro kontrolu vláken a programového běhu orbu*. Přitom mohou nastat nejrůznější běhové CORBA vyjímky, proto celý kód v 'main' uzavřeme v bloku 'try-catch' ošetřujícího CORBA výjimky. V bloku 'catch' obvykle minimálně zajistíme informativní výpisy, které přesměrujeme na výstupní zařízení, např. do chybového logu. Toto je obvykle nezbytné, pokud chceme mít jasný přehled o chybách v naší aplikaci. Toto jsou obvykle nejzákladnější akce, které bychom měli v metodě 'main' zajistit. Na přiloženém CD-ROM jsou zdrojové kódy aplikace CORBA-OPC můstku, proto zde je možné nalézt další informace.

3.2 Návrhové prostředí

3.2.1 Základní charakteristika, použité technologie

Návrhové prostředí pro vizualizaci bylo implementováno v jazyce Java, viz. např. [10] a [13]. Grafické uživatelské rozhraní bylo vytvořeno pomocí knihoven JFC/Swing. Pro některé výstupní soubory je použito formátu založeného na jazyce XML.

3.2.2 Instalace a spuštění

Aplikace návrhového prostředí se nijak zvlášť neinstaluje. Instalace se provádí jednoduše tak, že se celý obsah složky *soft/bin/np* přiloženého CD-ROM média překopíruje do požadované složky. Program se spustí skriptem *go.bat* umístěného ve složce. Ve skriptu *go.bat* musí být určena správná cesta k souboru *java.exe*, tj. k interpretu javy a k používaným knihovnám. Obsah skriptu *go.bat* by měl vypadat podobně jako v následujícím příkladu:

```
set JAVAPATH=c:\j2sdk1.4.1_02\bin
%JAVAPATH%\java -cp .\;.\lib\jdom.jar;.\lib\xerces.jar designer.DesignerMain
```

3.2.3 Popis návrhového prostředí

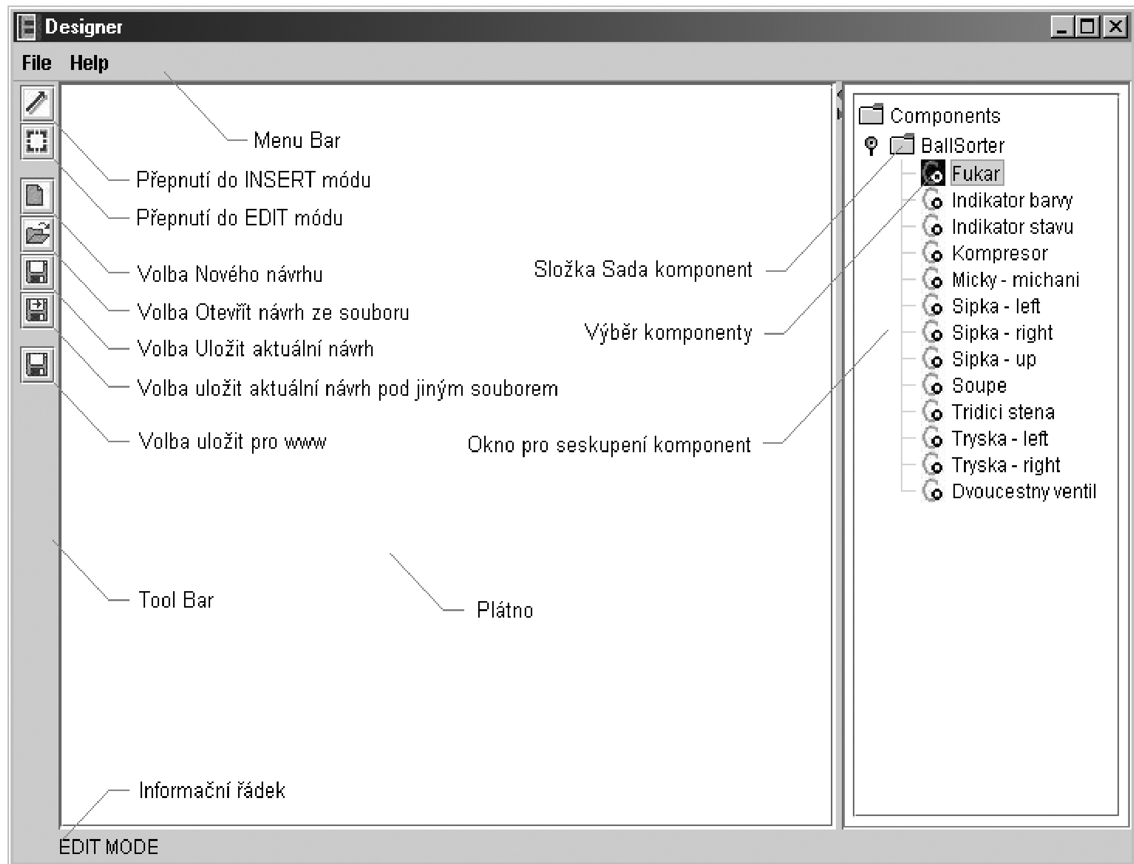
Podaří-li se spustit návrhové prostředí, objeví se hlavní okno, viz. obr. 16. Na bílém plátně uprostřed můžeme následně vytvářet návrh vizualizace. V pravém okně je místo pro seskupení komponent. Komponenty jsou organizovány ve stromu. Vlevo je umístěn 'Toolbar' s tlačítky pro různé funkce. Nahoře je hlavní menu aplikace, dole je umístěna informační lišta.

3.2.3.1 Práce v návrhovém prostředí

V návrhovém prostředí se pracuje s vizualizačními komponentami. Tyto komponenty představují ve vizualizaci obrazy skutečných objektů. Jednotlivé pracovní fáze při návrhu vizualizace je možné rozdělit následovně:

1. Vytvoření návrhu (nový, existující ze souboru)

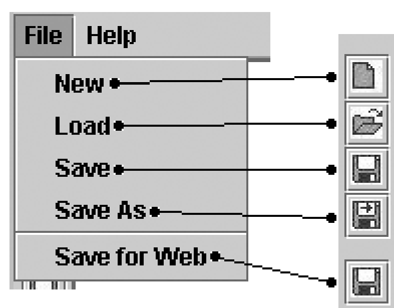
2. Práce s komponentami (přidávání komponent do návrhu, odstraňování, editace vlastností)
3. Uložení návrhu
4. Export pro www



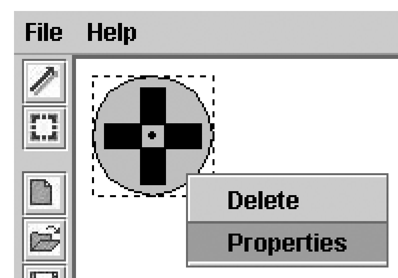
Obr. 16. Hlavní okno návrhového prostředí

3.2.3.1.1 Vytvoření návrhu

Nový návrh vytvoříme výběrem 'New' z menu 'File', obr. 17. Nový návrh se také automaticky vytvoří při spuštění návrhového prostředí. Rozpracovaný návrh je možné otevřít ze souboru, který jsem si předtím uložili. Návrhy se ukládají do formátu založeného na XML. Všechny tyto akce jsou dostupné prostřednictvím menu 'File'. Poté můžeme do návrhu přidávat a odstraňovat komponenty a měnit jejich vlastnosti.



Obr. 17: Menu 'File' a 'Toolbar'

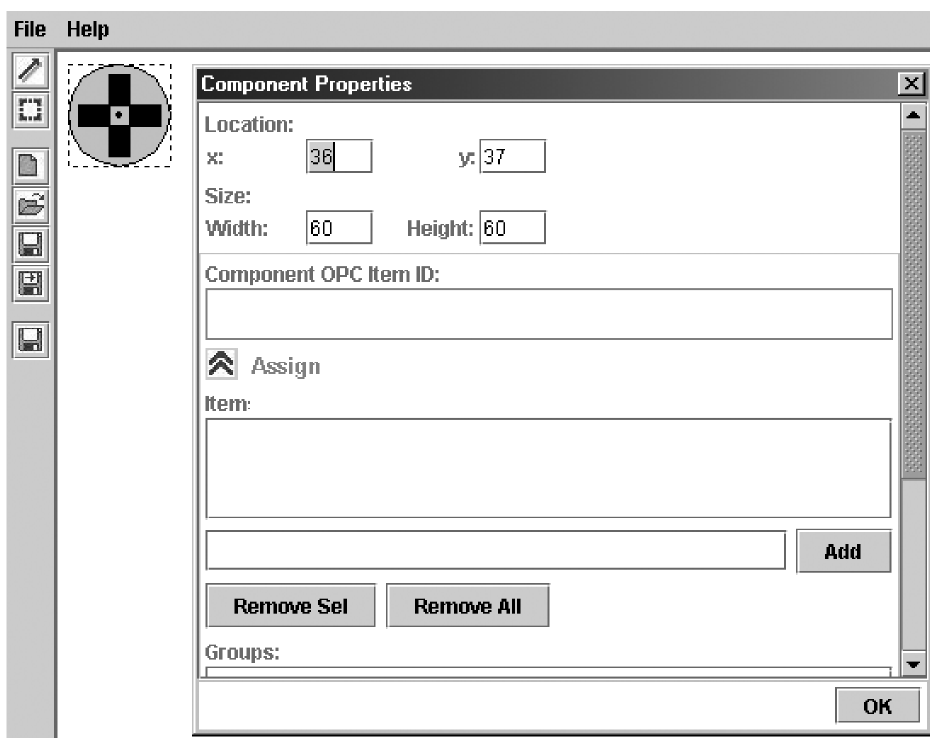


Obr. 18: Popup Menu

3.2.3.1.2 Práce s komponentami

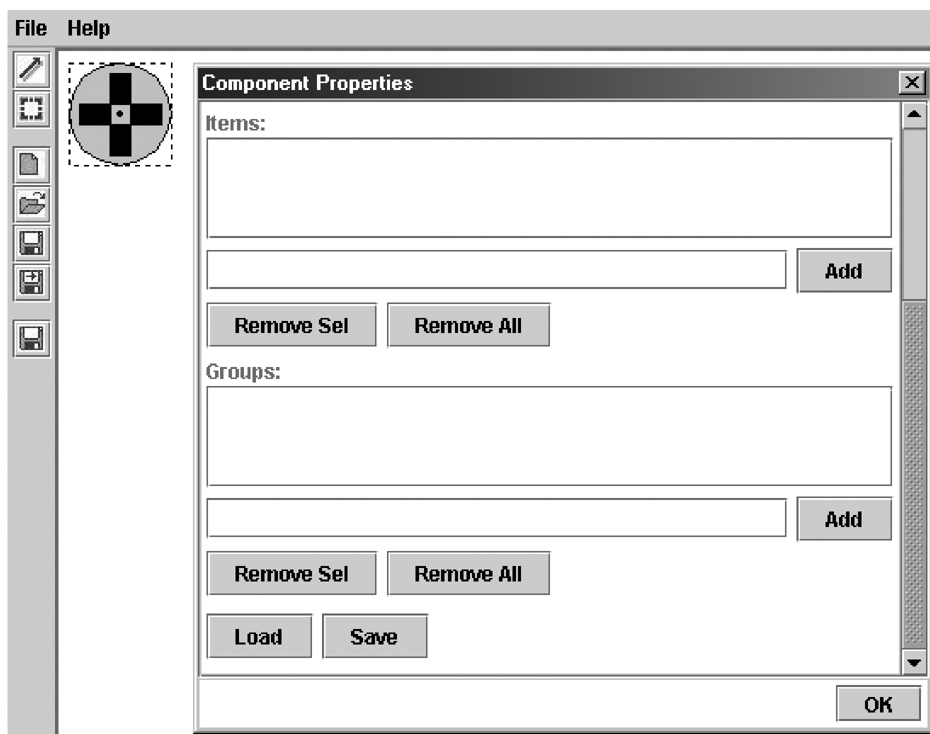
Chceme-li do návrhu přidat komponentu, přepneme se do vkládacího módu stiskem tlačítka 'Insert', obr. 16. Ve stromu komponent tlačítkem myši označíme požadovanou komponentu. Nyní se přesuneme na plátno a stiskem tlačítka umístíme komponentu.

Máme-li na plátně jednu nebo více komponent, můžeme měnit jejich vlastnosti. Přepneme se do editačního módu kliknutím na tlačítko 'Edit', viz. obr. 16. Označíme komponentu kliknutím na její obraz. Nyní můžeme kliknutím pravého tlačítka myši zobrazit 'Popup' Menu a vybrat požadovanou akci, viz. obr. 18. Zvolíme 'Delete' pro vymazání komponenty z návrhu. Zvolíme-li 'Properties', můžeme přiřadit nebo změnit komponentě některé vlastnosti. Zobrazí se okno 'Component Properties', viz. obr. 19.



Obr. 19: Editace vlastností komponent

Do textového pole pod oblastmi 'Items' a 'Groups' můžeme zadávat identifikátory OPC skupin a OPC itemů. Po zadání textového názvu klikneme na 'Add', tím se identifikátory zapíšou do seznamu. Seznam identifikátorů bude dostupný pro všechny komponenty v návrhu, proto je vhodné si seznam připravit předem a uložit si ho tlačítkem 'Save'. Seznam je možné nahrát ze souboru tlačítkem 'Load', obr. 20. Pokud chceme komponentě přiřadit OPC skupinu a OPC item, vybereme dvojici ze seznamu 'Items' a 'Groups' a klikneme na tlačítko 'Assign'. Poté co nastavíme vlastnosti pro komponentu, tlačítkem 'OK' zavřeme dialog.



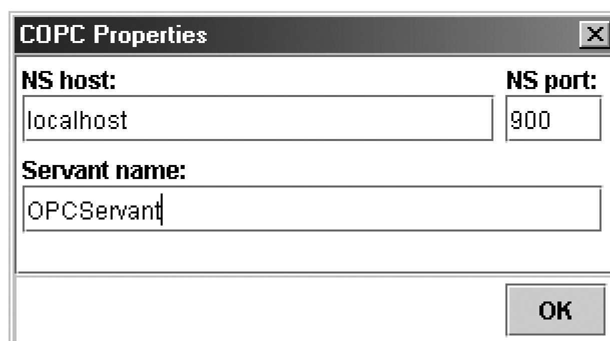
Obr. 20: Editace vlastností komponent

3.2.3.1.3 Uložení návrhu

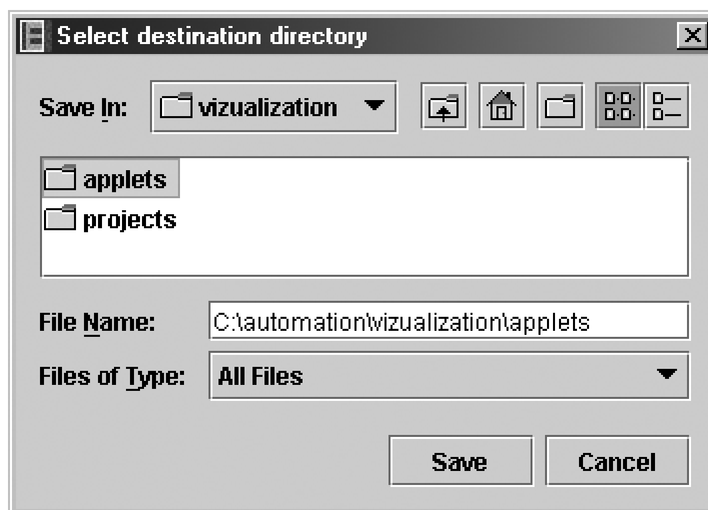
Návrh uložíme volbou 'Save' nebo 'Save As' z menu 'File', viz. obr. 17.

3.2.3.1.4 Export pro www

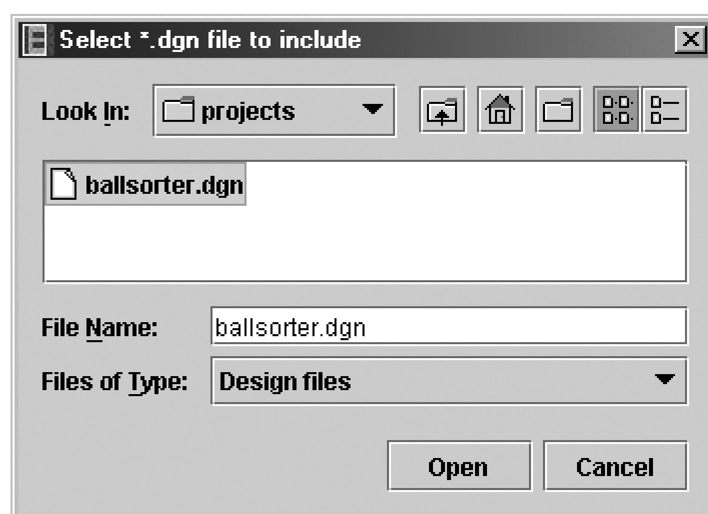
Poté, co jsme si uložili návrh vizualizace podle předchozího kroku, můžeme jej uložit pro www volbou 'Save for Web', viz. obr. 17. Pomocí této volby můžeme návrh zabalit do vizualizačního appletu. Výběrem této volby se postupně zobrazí okénka jak ukazují obr. 21, 22 a 23. V nich určíme parametry jmenné služby a jméno servanta C-OPC můstku ve jmenné službě. Následně vybereme složku pro export souborů a nakonec soubor návrhu, který chceme vyexportovat a který jsme si předtím uložili.



Obr. 21: Parametry C-OPC můstku



Obr. 22: Výběr cílové složky



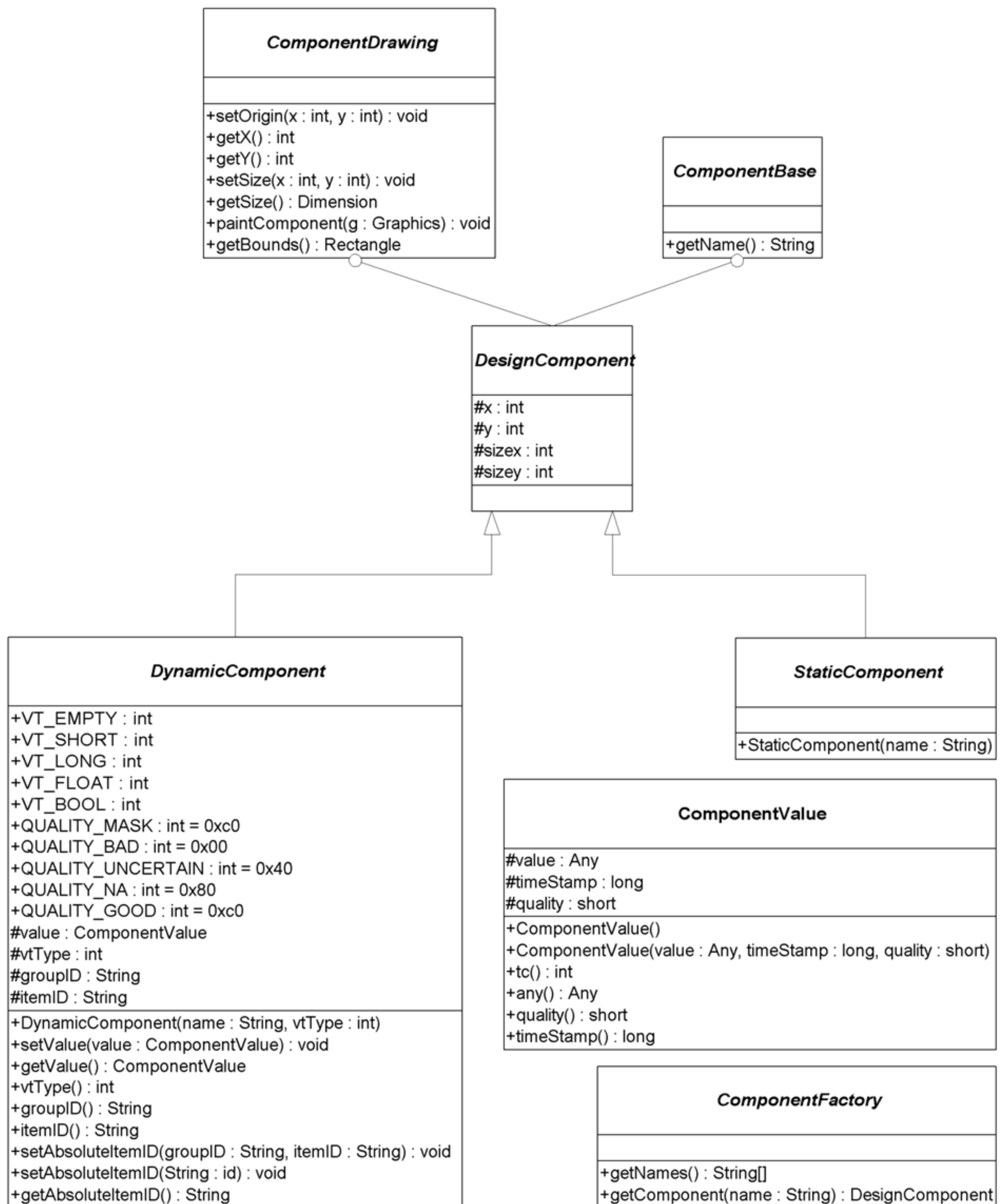
Obr. 23: Výběr souboru návrhu (vizualizační sestavy)

Výstupem je několik souborů (archív design.jar pro vlastní soubory vizualizačního appletu, archívy xerces.jar a jdom.jar a html dokument index.html, který je domovskou HTML stránkou appletu), které následně přemístíme (nejlépe do nového adresáře) do složky pro HTML dokumenty na WWW server. Tím se applet publikuje a bude dostupný v síti internet. Vizualizační applet se pak spustí ve WWW prohlížeči přechodem na jeho domovskou stránku.

Poznámka: Knihovny jdom.jar a xerces.jar jsou volně dostupné na www síti, viz. [14].

3.2.3.2 Vytváření komponenty

Návrhové prostředí nenabízí pomůcku pro vytváření vizualizačních komponent. Aplikaci je však možné jednoduše o tuto funkci rozšířit, protože aplikace byla vytvořena podle zásad objektového návrhu. Všechny komponenty vykazují určité společné rysy. Tyto rysy byly zahrnuty do společného aplikačního rozhraní komponent. Na obr. 24 vidíme hierarchii tříd komponent a jejich rozhraní.



Obr. 24: Hierarchie tříd a rozhraní vizualizační komponenty

Pro vytvoření komponenty však nemusíme nutně implementovat všechny metody rozhraní komponenty protože některé již byly implementovány. Pro začlenění nových komponent do návrhového prostředí je potřeba učinit následující kroky.

1. Implementujeme nové komponenty
2. Vytvoříme novou třídu implementující rozhraní ComponentFactory

3. Vytvoříme objekt nové třídy typu ComponentFactory a zaregistrujeme jej v aplikaci voláním metody setComponentFactory(ComponentFactory factory)
4. Aplikaci přeložíme

3.2.3.2.1 Implementace komponenty

a) Vybereme vhodného předka komponenty. V nejzákladnějším případě si vybereme abstraktní třídu DynamicComponent nebo abstraktní třídu StaticComponent. Třída DynamicComponent se od StaticComponent liší v tom, že umožňuje zachytit data OPC itemu. Podle hodnoty OPC itemu pak můžeme určit aktuální stav reálného objektu a tento stav zachytit v jeho obrazu, tj. ve vizuální podobě komponenty. Komponenta odvozená od StaticComponent hodnoty OPC nezachycuje a ve vizualizaci představují tyto komponenty pasivní komponenty, jejichž stav a vizuální podoba se nemění.

b) V odvozené třídě musíme definovat konstruktor, v němž jako první akci provedeme volání konstruktoru rodičovské třídy DynamicComponent nebo StaticComponent s požadovanými parametry. Parametry zachycují typ OPC dat, se kterými pracuje komponenta a jméno komponenty podle kterého budou dostupné aplikaci a proto musí být unikátní. OPC typy jsou deklarovány ve třídě DynamicComponent. Například, vizualizujeme-li stav zapnuto/vypnuto, stačí když určíme jako typ OPC dat typ DynamicComponent.VT_BOOL. Tento typ by měl reflektovat typ OPC itemu.

Poznámka: Při použití lomítka '/' v názvu komponenty se budou komponenty v návrhovém prostředí v panelu komponent zobrazovat ve stromu. Lomítka zde oddělují podobně jako adresáře jednotlivé úrovně stromu. Např. bude-li jméno komponenty "Ball Sorter/Fukar" uvidíme to v panelu komponent jak je vidět na obr. 25.

Obr. 25: Komponenta "Ball Sorter/Fukar" ve stromu komponent

c) Implementujeme metody *paintComponent()* a *getBounds()*.
Metoda *paintComponent()* přijímá jako parametr grafický kontext na nějž se má kreslit. Metoda *getBounds()* vrací objekt typu *Rectangle*, který definuje polohu a hranice vizuální podoby komponenty. Tato metoda slouží aplikaci, aby mohla určit interakce s komponentou při událostech od myši na plátně apod. Při implementaci komponent potřebujeme mít také určité znalosti o vytváření 2D grafiky v Javě. Pro informace o 2D grafice v Javě odkazuji např. na [5].

3.2.3.2.2 Třída ComponentFactory

Vytvoříme třídu implementující rozhraní *ComponentFactory*. V této třídě je potřeba implementovat metodu *getNames()* vracející jména dostupných komponent a metodu *getComponent(String name)* která vrací novou instanci komponenty podle daného jména komponenty.

3.2.3.2.3 Příklad

a) Implementace komponenty

```
// Semafor.java

package clib;

import mylib.*;
import java.awt.*;
import java.awt.geom.*;
import org.omg.CORBA.*;

public class Semafor extends DynamicComponent {
  public Semafor() {
    super("Semafor", DynamicComponent.VT_BOOL);
    sizex = 30;
    sizey = 30;
  }

  public void paintComponent(Graphics g) {
    Graphics2D g2d = (Graphics2D) g;
    g2d.translate(getX(), getY());

    g2d.setColor(Color.black);
    Rectangle obvod = new Rectangle(-sizex/2, -sizey/2, sizex, sizey);
    g2d.fill(obvod);
    Ellipse2D stred = new Ellipse2D.Double(-sizex/3, -sizey/3, 2*sizex/3,
2*sizey/3);

    if ((value != null) && (value.tc() == TCKind._tk_boolean)) {
      if (value.any().extract_boolean() == false) {
        // false
        g2d.setColor(Color.red);
        g2d.fill(stred);
      }
      else {
        // true
        g2d.setColor(Color.green);
        g2d.fill(stred);
      }
    }
    else {
      // out of data
    }

    g2d.translate(-getX(), -getY());
  }

  public Rectangle getBounds() {
    return(new Rectangle(getX() - sizex/2, getY() - sizey/2, sizex, sizey));
  }
}
```

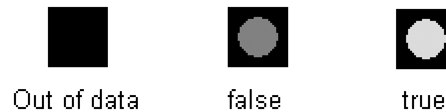
b) Implementace třídy typu ComponentFactory

```
// ComponentFactoryImpl.java

package clib;

import mylib.*;

public class ComponentFactoryImpl implements ComponentFactory {

  public String[] getNames() {
    String[] names = new String[1];
    names[0] = new Semafor().getName();
    return(names);
  }

  public DesignComponent getComponent(String name) {
    if (name == null) return(null);
    if (name.equals(new Semafor().getName())) return(new Semafor());
    return(null);
  }

}
```

c) Vytvoření a registrace instance třídy ComponentFactory

```
// DesignerMain.java
package designer;

import clib.*;
...

public class DesignerMain {
  public static void main(String[] args) {
    ...
    ComponentFactoryImpl componentFactory = new ComponentFactoryImpl();
    designer.setComponentFactory(componentFactory);
    ...
  }
}

// BrowserApplet.java

package applet.gui;

import clib.*;
...
public class BrowserApplet extends JApplet implements Runnable {
  public void init() {
    try {
      setComponentFactory(new ComponentFactoryImpl());
      ...
    }
    ...
}
```

Soubory Semafor.java a ComponentFactoryImpl.java přemístíme do adresáře *src/np/clib* a nakonec aplikaci přeložíme. Vizuální podoba komponenty Semafor z příkladu je vidět na obr. 26.



Obr. 26: Vizuální podoba komponenty z příkladu

3.2.3.3 Další možnosti vývoje

Jako další krok ve vývoji by bylo užitečné vytvořit pomocný nástroj pro návrh vizualizačních komponent. Tento nástroj by umožnil pohodlnější práci v návrhovém prostředí. Poté by bylo také užitečné vytvořit sadu všeobecně použitelných komponent dostupných v návrhovém prostředí, aby uživatel nemusel všechny komponenty vytvářet a mohl si vybrat z běžně používaných.

3.2.4 Překlad zdrojového kódu

Pro případ, že by bylo potřeba aplikaci znovu přeložit, např. pro novou verzi Javy apod. uvádím postup. V adresáři *soft/src/np*, kde se nachází zdrojové kódy, se nachází také skript *compile_idl.bat* a *compile_src.bat*. Pokud jsme provedli nějaké změny v souboru *copc.idl*, je potřeba jej znovu přeložit. K tomu slouží skript *compile_idl.bat*, ve kterém je potřeba určit cestu k IDL překladači *idlj.exe*:

```
// compile_idl.bat

set PATH=c:\j2sdk1.4.1_02\bin
idlj copc.idl
```

Poté je potřeba také přeložit zdrojové kódy jazyka Java, k tomu účelu jsem do složky přidal skript *compile_src.bat*. Soubory *options* a *paths* obsahují parametry překladu.

```
// compile_src.bat

set JAVACPATH=c:\j2sdk1.4.1_02\bin
%JAVACPATH%\javac -classpath .\lib\jdom.jar;.\lib\xerces.jar @options @paths
```

I když v Javě by měla být zachována zpětná kompatibilita mezi verzemi Javy, vždy je lepší aplikaci přeložit pro verzi Javy, která je nainstalovaná na počítači. Dále je potřeba si uvědomit, že beta verze často mohou obsahovat ještě neopravené chyby, proto je vždy lepší program přeložit a spouštět s použitím stabilní verze Javy.

3.3 Vizualizační applet

3.3.1 Základní charakteristika, použité technologie

Applet pro vizualizaci byl implementován v jazyce Java. Grafické uživatelské rozhraní bylo vytvořeno pomocí knihoven JFC/Swing. Pro některé vstupní soubory je použito formátu založeného na jazyce XML.

3.3.2 Úvod do problematiky appletu

Před tím, než vývojář začne psát applet, měl by se seznámit s principy psaní appletů a s tím, jak vlastně pracují a jaká skrývají úskalí. Proto jsem v této kapitole pro lepší pochopení vlastního návrhu shrnul základní a nejdůležitější poznatky o appletech.

3.3.2.1 Charakteristika appletu

Applet v javě je třída, resp. instance třídy odvozené od třídy *java.applet.Applet*. Applet představuje malý program, který je obvykle před spuštěním stažen ze sítě na hostující počítač a který obvykle běží v prostředí internetového prohlížeče. Protože na hostujícím počítači je applet pouze hostem, nemůže vykonávat činnosti obvyklé pro běžné applikace. Applet má například oproti aplikacím určitá bezpečnostní omezení. Od běžné aplikace se proto určitým způsobem odlišuje.

3.3.2.2 Životní cyklus appletu

Během existence appletu v prohlížeči applet prochází určitým cyklem, který určuje jeho momentální stav. Tento cyklus popisují čtyři metody appletu a to metoda *init()*, *start()*, *stop()* a *destroy()*. Tyto metody volá smyčka *JVM* zachycující události ve windows při výskytu určité specifické události. Těmito událostmi jsou nejčastěji otevření prohlížeče, reload prohlížeče, opuštění okna (přechod na jinou www stránku), návrat zpět na stránku appletu, maximalizace a minimalizace okna prohlížeče apod. To je nutné vzít v úvahu při implementaci appletu.

3.3.2.3 Konstruktor appletu

V appletu se obvykle nepoužívá konstruktor. Pokud je potřeba provést určité inicializace, je možné je provést v metodě *init()*. Jedním z důvodů je také to, že v konstruktoru appletu některé akce selhávají a v době provádění konstruktoru nemusí být ještě dostupné některé systémové prostředky (v konstruktoru obvykle např. selhává nahrávání obrázků do appletu).

3.3.2.4 Vykreslování appletu

Applet se vykresluje uvnitř metody *paint()* appletu. Tato metoda přijímá jako parametr grafický kontext, 'plátno' na nějž se má kreslit. Vývojář může tuto metodu překrýt, přičemž obvykle by měl jako první krok v této metodě volat rodičovskou metodu *paint()* rodičovské třídy appletu. Pak může následovat vlastní kreslení na 'plátno'. V každém případě by však vývojář neměl volat metodu *paint()* přímo. Mohl by se totiž dočkat nesprávného vykreslování, protože volání metody *paint()* by měl vykonávat pouze *layout manager* sám a také tak činí při určitých událostech (a vývojář by mohl

předávat nevyhovující plátno). Pokud však potřebuje vývojář naplánovat překreslení 'plátna', může volat metodu *repaint()*, která oznámí *layout manageru*, že má co nejdříve plátno překreslit (hned jak to bude možné a bude to mít smysl...když je například prohlížeč minimalizovaný a není nic vidět, *layout manager* pravděpodobně nemá nic překreslovat a měl by počkat až bude prohlížeč opět maximalizován).

3.3.2.5 Nahrávání a spouštění appletu v prohlížeči

Java applety je možné spouštět v různých aplikacích. Prakticky se jako běhové prostředí pro applety používá www prohlížeč do kterého se applet nahraje pomocí HTML stránky. Pro praktickou ukázku jak se do prohlížeče nahraje applet uvádím základní obsah takové stránky s komentářem:

```
// index.html

<html>
<head><title>Visualisation</title></head>
<body>
  <APPLET CODE="applet/gui/BrowserApplet.class"
ARCHIVE="design.jar,jdom.jar,xerces.jar" WIDTH="700" HEIGHT="800">
    <PARAM NAME="ORBInitRef"
VALUE="NameService=corbaloc::localhost:900/NameService">
  </APPLET>
</body>
</html>
```

HTML Tagem <APPLET> dáme prohlížeči příkaz k nahrání appletu do prohlížeče. Tento tag má atributy CODE, určující cestu ke třídě (souboru) appletu, tj. ke třídě, která dědí po *java.applet.Applet* a jejíž metoda *init()* se má volat. Atributem ARCHIVE je možné určit archívy *.jar (Java ARchive). Jednotlivá jména souborů *.jar se oddělují čárkou. Tyto komprimované soubory *.jar slouží k uložení nejrůznějších dat.

3.3.2.6 Archívy JAR

V archívech *.jar mohou být například java knihovny, používané appletem, stejně jako např. nejrůznější aplikační data. Archívy *.jar byly navrženy, aby se zredukoval objem dat přenášený s appletem po síti a tím také celková doba nahrávání appletu. Soubory *.jar používají stejnou metodu komprimace jako archívy *.zip (je proto možné jejich obsah zobrazit běžnými nástroji pro práci s archívy *.zip).

3.3.2.7 Předání parametrů appletu

Do appletu je také možné předat parametry viz. tag <PARAM>. Atributy tagu <PARAM> jsou identifikátor parametru NAME a hodnota parametru VALUE.

3.3.2.8 Bezpečnostní omezení

Při psaní appletu je nutné si také uvědomit, že některé akce není možné provést, protože jsou ze samotného principu bezpečnostní politiky pro applety zakázány. Prakticky to znamená, že některé metody z Java API nejsou vždy v appletech použitelné. Takové

metody obvykle vyhazují programovou vyjímku *java.lang.SecurityException*. Applety jsou pouze hosty na cílovém počítači a nemohou si dělat co chtějí.

Pro více informací o appletech odkazuji zájemce na [7].

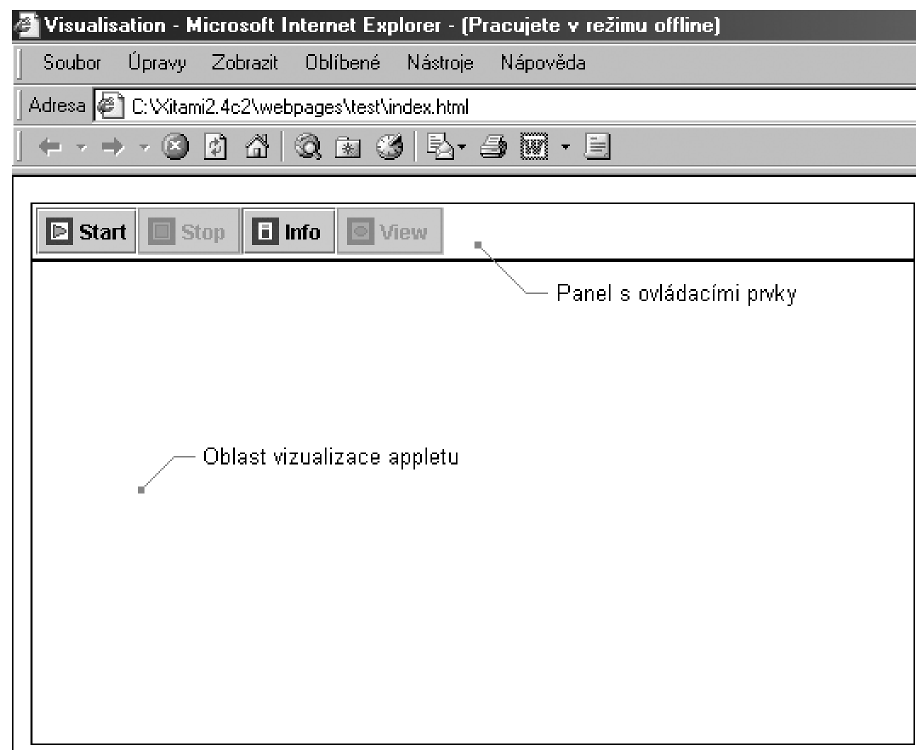
3.3.3 Podrobnosti implementace

Applet pro vizualizaci tvoří dvě vlákna. První vlákno obsluhuje uživatelské vstupy a druhé slouží ke čtení technologických dat ze sítě a zároveň vykreslování vizualizace na plátno. Pro samostatné vlákno pro uživatelské rozhraní jsem se rozhodl proto, aby případné výpadky sítě a čekání na data zbytečně neblokovaly uživatelské vstupy. Čtecí vlákno vykonává časově náročné operace.

Vlákna mezi sebou komunikují pomocí příznaků. Příznaky nastavuje vlákno obsluhující uživatelské vstupy. Druhé vlákno příznaky čte a podle jejich hodnoty koordinuje čtení technologických dat ze sítě a vykreslování vizualizace na plátno. Applet jsem implementoval podle zásad uvedených v odstavci *3.3.2 Úvod do problematiky appletu*. Applet navíc implementuje rozhraní *java.lang.Runnable*, které se v Javě používá k implementaci vláken. Metoda *run()* tohoto rozhraní implementuje čtecí vlákno. Další podrobnosti implementace je možné nalézt ve zdrojovém kódu na přiloženém CD-ROM médiu.

3.3.4 Popis prostředí appletu

Pokud při spouštění appletu nenastala nějaká závažná chyba, v prohlížeči by se mělo zobrazit uživatelské rozhraní appletu podle obr. 27.

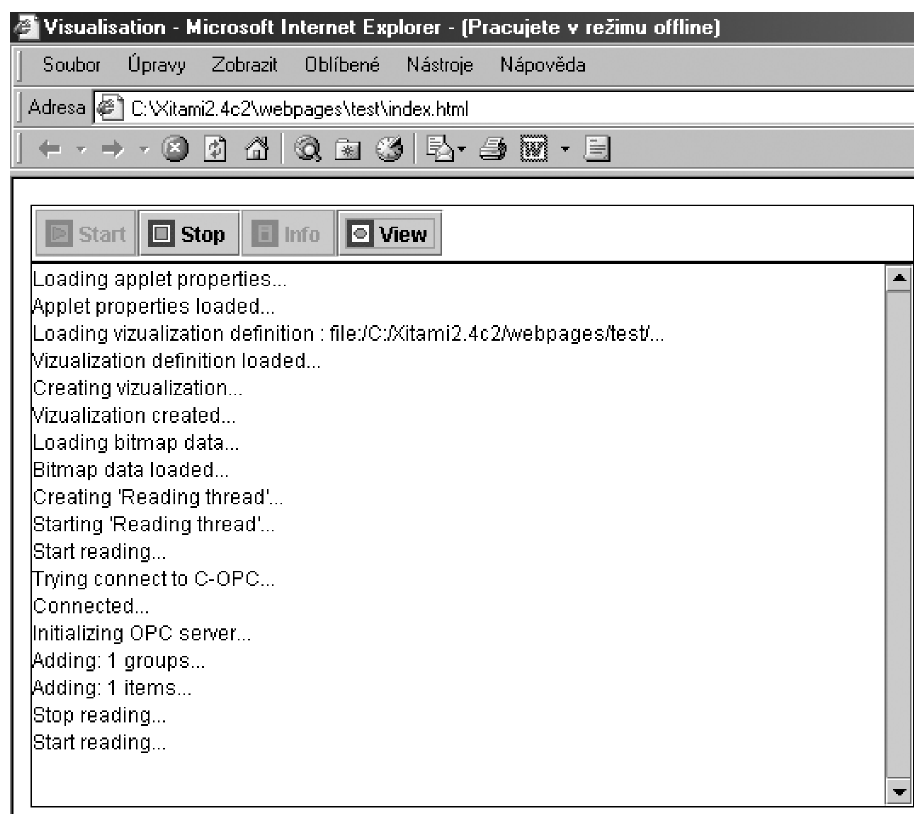


Obr. 27: Uživatelské rozhraní appletu

Uživatelské rozhraní tvoří panel s tlačítky pro jednoduché ovládání vizualizace, umístěný v horní části okna prohlížeče a pod tímto panelem orámovaná plocha pro zobrazení vizualizace. Pokud jsme v návrhovém prostředí vytvořili neprázdnou vizualizaci, měla by být i tato plocha neprázdná.

3.3.5 Ovládání appletu

Applet je možné ovládat aktivními tlačítky v horním panelu, které jsou zobrazeny barevně. Neaktivní tlačítka na uživatelské vstupy nereagují a jsou zobrazeny šedě. V počáteční fázi applet není k C-OPC připojen a barevně jsou zobrazeny tlačítka 'Start' a 'Info'. Kliknutím na tlačítko 'Start' dáme appletu pokyn k připojení k C-OPC serveru a k zahájení vizualizace. Nyní se tlačítko 'Start' zneaktivnilo a naopak se aktivovalo tlačítko 'Stop'. Tímto tlačítkem naopak pozastavíme vizualizaci. Kliknutím na 'Stop' se applet od C-OPC neodpojí, pouze pozastaví čtení dat ze sítě. Pokud však po delší dobu neklikneme opět na 'Start', C-OPC nás odpojí pro neaktivitu. Pokud se tak stane a dojdou vlákna na serveru, nebude možné ihned zahájit čtení, ale budeme muset čekat, až se na serveru nějaké vlákno uvolní. Kliknutím na tlačítko 'Info' můžeme monitorovat události vztahující se k fázi nahrávání a samotného běhu appletu. V orámované oblasti pro vizualizaci se zobrazí pracovní textové výpisy appletu. V nich je možné se dozvědět zda nějaká činnost neselhala. Výpis by mohl vypadat jak ukazuje obr. 28. Zpět k vizualizaci se dostaneme kliknutím na tlačítko 'View'. Detailnější, chybové výpisy můžeme také sledovat v Java konzoli.

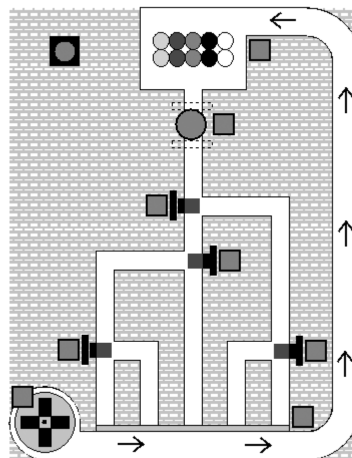


Obr. 28: Informační okénko appletu

3.4 Vizualizace třídicího mechanismu barevných míčků

Navržené prostředí pro vzdálenou vizualizaci, jehož popis byl obsahem této kapitoly, bylo potřeba nakonec otestovat na reálné úloze. Pro tento účel byla vybrána úloha z nabídky katedry řídicí techniky, a to řízení třídicího mechanismu barevných míčků. Pro tuto úlohu je v laboratoři katedry instalován reálný třídič, který je vybaven čidly a akčními členy pro jeho řízení a k němuž je připojeno PLC WAGO. Na této úloze si studenti procvičují praktický návrh řídicího algoritmu s použitím v průmyslu běžně používaných prostředků. Úloha je koncipována tak, aby umožnila vzdálené řešení úlohy. Za tímto účelem nají studenti k dispozici příslušný software pro vzdálené ovládání a rozhraní pro vizualizaci třídiče je dostupné na síti internet. V současnosti vizualizaci zajišťuje kamera, instalovaná v prostoru třídiče. Kamera umožňuje on-line pohled na třídič. Prostředí pro vzdálenou vizualizaci navržené v rámci diplomové práce by mělo pro tuto úlohu poskytnou doplňkový prostředek pro vizualizaci.

Pro vizualizaci třídícího mechanismu barevných míčků jsem vytvořil sadu jednoduchých komponent. S jejich pomocí byl v návrhovém prostředí vytvořen model třídiče, jenž můžeme vidět na obr. 29. Na obr. 30 je snímek třídiče pořízený z kamery umístěné v laboratoři. Následně byla provedena řada testů, které ověřily funkci PPVV.



Obr. 29: Vizualizace třídiče barevných míčků v PPVV

Obr. 30: Vizualizace třídiče pomocí webkamery

3.5 Obsah CD-ROM, organizace zdrojového kódu

3.5.1 Diplomová práce v elektronické podobě (doc, pdf)

Dokumenty diplomové práce v elektronické podobě ve formátech DOC a PDF jsou obsahem složky *text* na přiloženém CD-ROM.

3.5.2 Binární (spustitelný) kód aplikací

Aplikace CORBA-OPC můstku a návrhového prostředí (společně s appletem pro vizualizaci) byly přeloženy do spustitelné podoby. Popis jejich instalace a spouštění byl uveden v kapitolách 3.1.4 a 3.2.2. Aplikace CORBA-OPC můstku je umístěna ve složce *soft/bin/copc* a aplikace návrhového prostředí společně s appletem ve složce *soft/bin/np*.

3.5.3 Zdrojový kód aplikací

Zdrojový kód vytvořených aplikací je obsahem přiloženého CD-ROM. Pro popis toho, co která složka obsahuje, viz. tab. 12.

Tab. 12: Obsah přiloženého CD-ROM

Složka	Popis
soft/src/copc	zdrojové kódy COPC můstku
soft/src/np	zdrojové kódy appletu a vývojového prostředí
np/applet	třídy, které používá pouze applet
np/designer	třídy, které používá pouze vývojové prostředí
np/clib	implementace vizualizačních komponent
np/copc	třídy generované idl kompilátorem (COPC stub, objekty typů deklarovaných v copc.idl, IOPC rozhraní ...)
np/component	třídy a rozhraní pro vizualizační komponenty (ne implementace)
np/lib	externí knihovny (jdom.jar, xerces.jar)
np/xml	pomocné třídy pro práci s XML
np/applet/config	různé proměnné (relativní cesty, XML tagy...)
np/applet/gui	třídy uřivatelského rozhraní
np/applet/icons	soubory bitmapových obrázků (JPG)
np/applet/util	třídy definující různé pomocné funkce
np/designer/config	různé proměnné (relativní cesty, XML tagy...)
np/designer/gui	třídy uřivatelského rozhraní
np/designer/icons	soubory bitmapových obrázků (JPG)

4 Závěr

V přípravě na diplomovou práci jsem nastudoval technologii *corba*. V této fázi bylo potřeba zpracovat mnoho stránek specifikace. Výsledek této studie jsem zahrnul to teoretické části diplomové práce. Ačkoli text zdaleka neobsahuje všechny detaily *corby*, věřím, že poslouží mnoha zájemcům o tuto technologii. Snažil jsem se v ní shrnout nejpoužívanější rysy a také rysy, které jsem považoval za důležité pro pochopení celku. Sám jsem se v této fázi práce o *corbě* mnoho naučil. Současně bylo potřeba nastudovat také technologie COM a OPC. Nabyté znalosti jsem následně využil při návrhu CORBA-OPC můstku.

V druhé fázi práce jsem přistoupil k vlastní realizaci. Implementace můstku se víceméně dostala do předpokládané podoby, ačkoli můstek nemá implementovány všechny metody OPC rozhraní. OPC rozhraní je poměrně rozsáhlé, jeho kompletní implementace v C-OPC můstku se do vyčleněného časového horizontu práce nevešla. Z těch nejzákladnějších například neimplementuje metody zápisu na OPC server. Tyto funkce nejsou nezbytné pro vizualizaci. Rozšířením můstku o tyto metody a odpovídajícího aplikačního rozšíření bychom však dospěli o další krok kupředu a získali větší možnosti použití, kupř. pro vzdálené řízení či ovládání. Můstek je na základní úrovni konfigurovatelný, jako server podporuje paralelní zpracování klientských požadavků a implementuje funkci timeoutu. Pro nastavení optimální výpočetní zátěže umožňuje nastavit maximální počet vláken zpracovávajících požadavky.

Základním úkolem bylo také vyzkoušet můstek v praxi. Činnost můstku byla ověřena v aplikaci vizualizace technologického procesu. Za tímto účelem jsem implementoval Java applet, který implementuje funkci CORBA klienta a umožňuje interpretovat a vizualizovat technologická data dostupná prostřednictvím můstku ze zdroje dat - OPC serveru. Applet poskytuje jednoduché uživatelské rozhraní pro snadné ovládání vizualizace a připojení ke zdroji dat a informuje uživatele o kritických běhových stavech.

Pro podporu návrhu vizualizace jsem implementoval jednoduché návrhové prostředí. Návrhové prostředí umožňuje vytvářet vizualizační sestavy kompozicí dostupných komponent. Kromě základních funkcí umožňuje exportovat vizualizační sestavy pro www.

V dalších fázích vývoje návrhového prostředí bude vhodné vytvořit pomocný nástroj pro návrh vizualizačních komponent. Ten bude možné do vývojového prostředí snadno začlenit podle postupu který byl zmíněn v odstavci 3.2.3.2.

V rámci diplomové práce se také podařilo prověřit možnosti propojení dvou nezávislých softwarových technologií CORBA a COM. Obě technologie byly vytvořeny pro podobné účely a cíle, avšak s odlišným přístupem vývojářů. Ukázalo se, že spolupráce těchto dvou technologií je možná a navržené aplikace poukazují, že je možné současně využít výhody obou technologií. To významně rozšiřuje použitelnost stávajících COM a CORBA aplikací a umožňuje dosáhnout jejich součinnosti.

5 Reference

Literatura

[1] Common Object Request Broker Architecture, Object Management Group, 2002

[2] Naming Service Specification, Object Management Group, 2002

[3] Visibroker for C++ 4.0 Reference, Inprise, 2000

[4] Visibroker for C++ 4.0 Programmers guide, Inprise, 2000

[5] The Java Tutorial: 2D Graphics, Sun Microsystems, 2001

[6] Tomáš Svoboda, Průmyslová automatizace s využitím OPC, Diplomová práce, ČVUT v Praze, 2003

[7] The Java Tutorial: Writing Applets, Sun Microsystems, 2001

[8] C++ Language Mapping Specification, Object Management Group, 1999

[9] IDL to Java Language Mapping Specification, Object Management Group, 2002

[10] Pavel Herout, Učebnice jazyka Java, Nakladatelství Kopp, České Budějovice, 2000

Internet

[11] http://www.omg.org

[12] http://www.opcfoundation.org

[13] http://www.javasoft.com

[14] http://www.jdom.org

[15] http://www.huihoo.com/corba/free_corba.html