České vysoké učení technické v Praze

fakulta elektrotechnická

katedra řídicí techniky

Diplomová práce

Průmyslová automatizace s využitím OPC

Tomáš Svoboda

2003

Zadávací formulář

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW 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

Tato diplomová práce se zabývá vývojem serveru pro zpracování historických dat v souladu se specifikací OPC Historical Data Access 1.0. Z nepovinných částí této specifikace je podporováno pouze rozhraní pro synchronní čtení dat. Ostatní volitelná rozhraní nejsou implementována. Databáze historických dat je vytvořena pod databázovým serverem MySQL. Vytvořen byl rovněž univerzální klient pro grafickou interpretaci dat poskytovaných OPC HDA servery a nástroje pro záznam procesních dat do databáze.

V teoretické části práce je uveden stručný popis technologie COM firmy Microsoft, použitých OPC specifikací a struktury vytvořené databáze. Zmíněna je rovněž teorie návrhu databází, popis databázového serveru MySQL, způsob komunikace s databázovým serverem a popis vytvořených aplikací.

Annotation

This thesis describes the implementation of historical data server according to OPC Historical Data Access 1.0 specification. Only the interface for synchronous data reading is implemented from the optional part of this specification. Database for storing the historical data is created under MySQL database server. There is also implemented the client for graphical representation of data provided by OPC HDA servers and tools for recording process data to the database.

Theoretical part contains brief description of Microsoft COM technology, used OPC specifications and the structure of the database for process data. Some theory of database design, the way of communication with database server and description of created applications is also mentioned.

Obsah

ZADÁVACÍ FORMULÁŘ .... **I**

PROHLÁŠENÍ .... **II**

ANOTACE .... **III**

ANNOTATION .... **III**

OBSAH .... **IV**

1 ÚVOD **1**

2 TECHNOLOGIE COM **3**

2.1 TECHNOLOGIE KOMPONENTOVÝCH APLIKACÍ 3

2.1.1 Komponenty, rozhraní a metody *3*

2.1.2 Identifikátory GUID, CLSID, IID *5*

2.1.3 Marshalling *5*

2.1.4 Jazyk IDL *6*

2.1.5 Rozhraní IUnknown *7*

2.1.6 Vytváření objektů a ClassObject *8*

2.1.7 Agregace objektů *11*

2.1.8 Rozšíření základního modelu COM *11*

2.1.9 Typové knihovny *12*

2.1.10 IDispatch a Dual rozhraní *13*

2.1.11 Threading model *15*

2.2 STRUKTURA COM APLIKACE 16

2.2.1 Alokace paměti pro přenášené parametry *16*

2.2.2 COM Server *17*

2.2.3 Connection Point *18*

3 STANDARDY OPC **20**

3.1 OBECNÉ INFORMACE 20

3.1.1 Důvod vzniku *20*

3.1.2 Typy OPC specifikací *21*

3.1.3 Architektura OPC aplikací *23*

3.1.4 Custom a Automation rozhraní *24*

3.1.5 Rozhraní IOPCCommon *25*

3.1.6 Rozhraní IOPCShutdown *25*

3.1.7 OPC Server Browser *26*

3.2 DATA ACCESS 2.0 26

3.2.1 Účel specifikace 26

3.2.2 Přehled objektů 27

3.2.3 Rozhraní objektu OPC Server 29

3.2.4 Rozhraní objektu OPC Group 30

3.2.5 IOPCDataCallBack 31

3.3 HISTORICAL DATA ACCESS 1.0 32

3.3.1 Účel specifikace 32

3.3.2 Přehled objektů 33

3.3.3 Rozhraní objektu Server 34

3.3.4 Metody čtení dat 35

3.3.5 Agregační funkce 36

4 DATABÁZE PROCESNÍCH DAT 37

4.1.1 Použitý DB Server a připojení k serveru 37

4.1.2 Požadavky na databázi 38

4.1.3 Struktura databáze 40

4.1.4 Ukládání časových značek 43

4.1.5 Definice a metoda výpočtu agregačních funkcí 45

5 POPIS APLIKACÍ 47

5.1 OPC CLIENT 47

5.1.1 Objekty pro OPC DA 47

5.1.2 Grafická část aplikace 50

5.2 HDA SERVER 51

5.2.1 Struktura aplikace 51

5.2.2 Třída CoOPCHDAServerImpl 53

5.2.3 Třída CoOPCHDABrowserImpl 54

5.2.4 Konfigurace databáze 54

5.3 KOMPONENTA HDA CLIENT 55

5.4 HDA CLIENT 58

5.5 DB OPC CLIENT 59

5.6 DB UNI CLIENT 61

5.7 ACTIVEX HDA CLIENT 61

6 ZÁVĚR 62

7 SEZNAM POUŽITÉ LITERATURY 64

8 PŘÍLOHY 65

PŘÍLOHA A - SQL DEFINICE TABULEK 65

PŘÍLOHA B - DEFINOVANÉ ATRIBUTY POLOŽEK 66

PŘÍLOHA C - DEFINOVANÉ AGREGAČNÍ FUNKCE 66
PŘÍLOHA D - INSTALACE A POPIS OVLÁDÁNÍ VYTVOŘENÝCH PROGRAMŮ 68
Instalace programů *68*
Ovládání programu OPC Client *69*
Ovládání programu HDA Server a DB Config *70*
Ovládání programu HDA Client *71*
Ovládání programu DB OPC Client *74*
PŘÍLOHA E - WWW SERVERY SOUVISEJÍCÍ S UVEDENOU TÉMATIKOU 75
PŘÍLOHA F – UKÁZKA DEFINICE ADRESOVÉHO PROSTORU SERVERU 76

1 Úvod

Využití řídicích systémů v průmyslu se neustále rozrůstá. S nárůstem automatizace řídicích procesů narůstá i množství dat jak o řízeném procesu, tak o samotných zařízeních. Tato data musí být dostupná klientům na všech úrovních informační architektury. Do nedávné doby používalo mnoho klientů zpracovávajících data z technologií své vlastní komunikační prostředky v podobě specializovaných ovladačů pro dané zařízení. To s sebou přinášelo mnoho nevýhod. Mezi největší z nich patří:

- nutnost vyvíjet vždy nové ovladače pro každé zařízení a tak vznikající zbytečná duplicita,
- neslučitelnost ovladačů od různých výrobců díky různým podporovaným hardwarovým funkcím,
- konflikty v přístupu k zařízením. Dvě různé aplikace většinou nemohly přistupovat najednou k jednomu zařízení, protože používaly různé ovladače.

Výrobci zařízení se snažili čelit těmto problém pomocí vývoje univerzálních ovladačů. Naráželi však na potíže způsobené různými požadavky klientů. Není prakticky možné vyvinout jediný ovladač schopný spolupracovat se všemi klienty.

Z uvedeného vyplývá nutnost zavedení jednotného způsobu výměny dat mezi zařízeními a softwarovými klienty zpracovávajícími data z těchto zařízení. Z těchto důvodů vznikla mezinárodní organizace OPC Foundation, jejímž hlavním cílem je vývoj jednotných mechanismů pro komunikaci s datovými zdroji, ať už se jedná o zařízení na úrovni řízení samotné technologie nebo o zdroje na vyšší úrovni informační architektury, například databází. Výsledkem je soubor specifikací, označených souhrnně zkratkou OPC (OLE for Process Control), které definují způsoby komunikace v různých oblastech řídicí techniky.

Díky velkému objemu dat automaticky dostupných z řídicích zařízení představují databáze historických dat důležitý zdroj informací. Záznam historických dat umožňuje ucelený pohled na činnost řídicího systému v libovolném časovém období. Umožňuje jak analýzu a optimalizaci řízeného procesu a řídicích algoritmů, tak i predikci dalšího vývoje a plánování optimálních budoucích strategií. Informace poskytované historickými databázemi musí být proto dostupné pro všechny koncové uživatele a klienty, kteří je mohou využít. Z těchto důvodů vznikla specifikace OPC Historical Data Acsess, jejímž účelem je sjednotit způsob výměny dat mezi servery pro historická data a klienty zpracovávajícími tato data. Vzniklo tak otevřené komunikační prostředí, ve kterém mohou být použity aplikace různých výrobců. Cílem praktické

části této diplomové práce je vytvořit server pro práci s historickými daty uloženými ve vytvořené databázi, který vyhoví uvedené specifikaci. Rovněž bylo nutné vytvořit nástroje pro záznam dat do databáze a implementovat klient spolupracující s HDA servery. Klient umožňuje grafickou reprezentaci poskytovaných dat.

Základem, na kterém jsou postaveny všechny OPC specifikace, je technologie COM (Common Object Model) vyvinutá firmou Microsoft a implementovaná do všech moderních operačních systémů této firmy. Jedná se o technologii umožňující vytváření objektů se specifikovanou funkcí použitelných univerzálně v kterékoliv cílové aplikaci. V rámci COM jsou specifikována jak pravidla pro vytváření těchto objektů, tak i způsob komunikace mezi klientskou aplikací a objektem. Protože porozumění této technologii je nutné pro pochopení principů OPC specifikací, začneme teoretickou část této práce jejím popisem COM v kapitole 2.

Obecný úvod do filosofie OPC a podrobnější rozbor dvou specifikací použitých pro vypracování praktické části této práce je uveden v kapitole 3. Jako první se zabýváme specifikací OPC Data Access, která je určena především pro výměnu dat mezi aplikacemi a technologií v reálném čase a je tudíž vhodná pro sběr dat do databází historických dat. Jako druhá je popsána již zmíněná Historical Data Access popisující rozhraní a funkce HDA serveru.

Nedílnou součástí každého serveru pro historická data je samozřejmě databáze, ve které jsou sledovaná data uložena. V kapitole 4 se zabýváme způsobem připojení k databázi a popisem použitého databázového serveru. Je zde také zmíněna teorie návrhu databáze pro uložení procesních dat a její konkrétní struktura, použitá pro účely této diplomové práce.

Popis jednotlivých aplikací vytvořených v praktické části je uveden v kapitole 5. Jsou zde uvedeny především informace o účelu aplikací a jejich struktuře. Popsány jsou také funkce důležitých objektů implementovaných v programech. V příloze je pak možné nalézt podrobnější informace o funkci aplikací a o způsobu jejich ovládání.

2 Technologie COM

COM je zkratka pro *Common Object Model,* což je standard, který umožňuje vyvíjet univerzální, na platformě nezávislé komponenty použitelné prakticky ve všech jazycích a vývojových prostředích. Za jeho předchůdce je považován standard OLE (*Object Linking and Embedding)*, vyvinutý firmou Microsoft, který umožňoval používání dokumentů jednotlivých aplikací v dokumentech jiných aplikací.

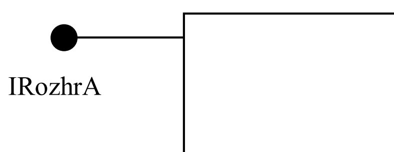
Samotná technologie COM se objevila v roce 1995. Její rozšíření je spojené především s operačními systémy Windows firmy Microsoft, ale existují metody, které umožní použití i komponent z jiných operačních systémů (např. UNIX).

Díky použití DCOM (*Distributed COM*) je možné vytvářet i distribuované komponentové aplikace. Pro uživatele je umístění komponenty transparentní a nezáleží tedy na tom, zda ji používá jako in-process, lokální, nebo vzdálenou.

2.1 Technologie komponentových aplikací

2.1.1 Komponenty, rozhraní a metody

Komponenta je základním stavebním prvkem COM aplikace. Je to objekt s přesně definovanou funkcí, který je možné díky COM použít prakticky kdekoliv. Jednou ze základních vlastností komponent je jejich dokonalé zapouzdření. Uživatel nemá žádné informace o tom jaká je jejich vnitřní struktura, pouze o jejich funkci. Z tohoto důvodu jsou často komponenty znázorňovány jako jakési černé skříňky. Komunikace mezi aplikací a komponentou ale musí být z důvodů její univerzálnosti samozřejmě standardizována. K tomu slouží tzv. rozhraní (*Interface*). Ty představují brány, skrz které je jediný možný přístup ke komponentě. Schematicky je pak možné komponentu zobrazit jako na obr. 2.1



obr. 2.1 Znázornění objektu a jeho rozhraní

Rozhraní představuje jednak seznam funkcí, které může klient volat, ale především definuje chování komponenty při volání jednotlivých funkcí. Je to tedy jakýsi kontrakt mezi klientem a komponentou. Tento kontrakt musí být pro všechny verze komponenty neměnný. Pokud je zapotřebí definovat nové chování, je nutné definovat nové rozhraní. Tím je zajištěna

zpětná kompatibilita různých verzí. Každá komponenta může implementovat libovolný počet rozhraní.

Rozhraní jsou v jazyce C++ reprezentována jako abstraktní třídy s virtuálními funkcemi. Pro příklad z obr. 2.1 by pak mohla taková definice vypadat například takto:

```
interface IRozhrA : public IUnknown
{
  virtual HRESULT STDMETHODCALLTYPE Func1(void) = 0;
  virtual HRESULT STDMETHODCALLTYPE Func2(int Count) = 0;
  virtual HRESULT STDMETHODCALLTYPE Func3(int *Count) = 0;
}
```

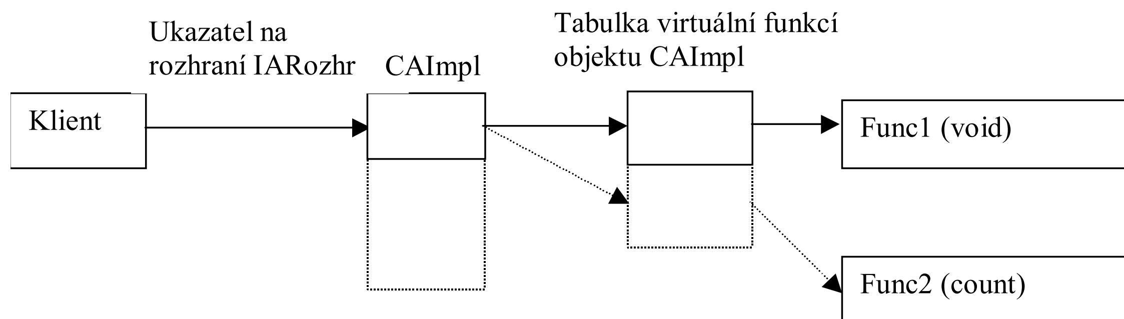
Identifikátor interface není v jazyce C++ klíčové slovo, ale je definován jako struct. Podle konvence začínají jména rozhraní písmenem I. Všechna rozhraní dědí od IUnknown, což je základní typ rozhraní povinný pro všechny komponenty. Jeho bližší popis je uveden v kapitole 2.1.5. Z uvedeného příkladu je vidět, že definované rozhraní obsahuje tři funkce (*Func1* až *Func3*), které tak jako prakticky všechny funkce rozhraní COM, mají návratový typ HRESULT. Jedná se o 32-bitové číslo představující chybový kód. Znaménkový bit je použit pro rozlišení úspěchu a neúspěchu prováděné operace. Zbývající bity pak mohou poskytnout detailnější informaci.

Druhou datovou strukturou komponenty je třída, které realizuje vlastní implementaci Implementace obsahuje členská data, metody a především implementaci všech definovaných metod všech rozhraní. Třída implementující dané rozhraní vždy dědí od tohoto rozhraní. Pro výše uvedený příklad tedy můžeme navrhnout například následující implementační třídu:

```
Class CAImpl : public IrozhrA
{
  HRESULT STDMETHODCALLTYPE Func1(void);
  HRESULT STDMETHODCALLTYPE Func2(int Count);
  HRESULT STDMETHODCALLTYPE Func3(int *Count);
}
```

Volání funkcí rozhraní klientem (obr. 2.2) je pak shodné s voláním virtuálních funkcí v jazyce C. Jedinou informaci o komponentě, kterou má klient k dispozici, je ukazatel na její rozhraní. Při volání metody rozhraní se provádí několik operací: nejdříve je provedena dereference ukazatele na rozhraní a nalezen ukazatel na tabulku virtuálních funkcí (*VTable*). Na

získaném ukazateli je opět provedena dereference a indexováním do tabulky je získána skutečná adresa funkce, která je volána.



obr. 2.2 Schéma volání funkcí rozhraní

2.1.2 Identifikátory GUID, CLSID, IID

Protože všechny objekty vytvořené v souladu s požadavky COM musí být univerzálně použitelné, je nutné mít k dispozici identifikátory, které budou jednotlivé komponenty a jejich rozhraní jednoznačně určovat. Tyto identifikátory musí být stejné pro všechny operační systémy a počítače, na kterých se budou komponenty používat. Pro tento účel se používají 128 bitová čísla nazývaná obecně GUID (Globally Unique IDentifier). Použití takto rozsáhlé struktury zajistí dostatečné množství identifikátorů pro všechny komponenty.

Pro generaci GUID se používají specielní algoritmy, které s použitím aktuálního data a času, unikátního čísla síťové karty a dalších údajů zajistí neopakovatelnost těchto identifikátorů v globálním měřítku. Tyto algoritmy jsou implementovány v programu *guidgen.exe* a generace GUID je zajištěna automaticky vývojovými nástroji.

V jazyce C++ jsou nadefinovány ještě další datové typy se shodnou strukturou. Pro identifikaci tříd (komponent) se používá CLSID (Class Identifier), pro rozhraní pak IID (Interface Identifier).

V systémech Windows jsou informace o každém zaregistrovaném objektu COM uloženy v registrech. Spolu s GUID daného objektu jsou zde uloženy údaje o umístění příslušného souboru (DLL, EXE, OCX), textový identifikátor objektu (ProgID) a další informace.

2.1.3 Marshalling

Komponenty COM mohou pracovat jako in-process, lokální nebo vzdálené servery. Komunikace mezi komponentou a klientskou aplikací se odehrává výhradně pomocí volání metod rozhraní, které komponenta implementuje. Pro výměnu dat mezi komponentou a klientem

se předává ukazatel na tato data v paměti. Pro komponenty pracující jako in-process funguje vše stejně jako v případě klasického volání funkcí v rámci jedné aplikace, protože komponenta sdílí s klientskou aplikací stejný adresový prostor. Lokální a vzdálené komponenty však mají svůj vlastní adresový prostor a proto je pro výměnu dat potřeba složitějšího procesu. Ten se nazývá marshalling dat. Jedná se vlastně o kopírování dat mezi adresovými prostory. Marshalling musí být zajištěn implementací COM na daném systému, protože pro klientskou aplikaci musí být umístění komponenty transparentní.

V prostředí Windows je zajištěn automatický marshalling pro všechny předávané parametry jejichž typ spadá do skupiny *automation compatible* typů. Podrobný výčet této skupiny je možné nalézt například v [1].

Pro všechny ostatní typy předávaných dat musí marshalling zajistit tvůrce komponenty. Děje se tak pomocí tzv. proxy/stub knihoven, které je možné automaticky vytvořit pomocí kompilátoru jazyka IDL (2.1.4). Proxy knihovna pracuje na straně klienta. Při volání jakýchkoliv funkcí rozhraní uloží klient předávané argumenty do zásobníku. Pokud funkce není in-process, je volání předáno do proxy. Ta zajistí sbalení parametrů do marshalling packetu a jeho přenos k vzdálenému objektu. Tam je za pomoci stub knihovny packet rozbalen, parametry uloženy do zásobníku a znovu vytvořeno volání funkce v místním adresovém prostoru.

Proxy a stub knihovny zajišťují také zpřístupnění ukazatelů na funkce rozhraní v adresovém prostoru klienta.

2.1.4 Jazyk IDL

Protože COM musí zajistit marshalling dat mezi komponentou a klientskou aplikací, nestačí pro popis rozhraní pouhý výčet metod a datových typů jejich parametrů. Je potřeba, aby systém měl informace také o tom, zda se jedná o parametry vstupní nebo výstupní a v případě polí i o jejich velikosti. Z těchto důvodů se pro popis rozhraní používá jazyk IDL.

Jazyk IDL byl původně navržen organizací Open Software Foundation a jeho účelem byl popis používaných datových typů a funkcí spouštěných mimo počítač klienta. Firma Microsoft jej převzala a rozšířila tak, aby se dal použít k popisu rozhraní COM komponent.

Popis dané třídy a jejích rozhraní pomocí IDL dává systému všechny potřebné informace pro zajištění komunikace mezi komponentou a klientem.

Jako příklad uvedeme popis rozhraní z kapitoly 2.1.1.

```
[object, uuid(E312522F-A7B7-A52E-0000F8751BA7]
interface IRozhrA : IUnknown
{
    HRESULT Func1();
    HRESULT Func2([in] int Count);
    HRESULT Func3([in, out] int *Count);
}
```

Definice třídy CAImpl implementující uvedené rozhraní by pak vypadala následovně:

```
[object, uuid(E312522E-A7B7-A52E-0000F8751BA7]
coclass CAImpl
{
    [default] interface IRozhrA;
}
```

V hranatých závorkách jsou u každého prvku uvedeny příslušné atributy. Pro třídu nebo rozhraní je to především jejich GUID a klíčové slovo objekt. Pro parametry metod rozhraní jsou definovány následující atributy:

[in]	Data jsou naplněna volající stranou (klientem) a jsou přenesena pouze v jednom směru
[out]	Data jsou naplněna komponentou a přenášena od komponenty k volající straně. V opačném směru je předán pouze ukazatel.
[in, out]	Parametr je přenášen oběma směry
[out, retval]	Při použití atributu retval představuje pak daný parametr pro klienty některých typů (Visual Basic, Java) návratovou hodnotu funkce.
[size_is Vel]	Parametr Vel určuje počet prvků pole daného parametru.

Popis datových struktur a konstant v jazyce IDL je totožný s jazykem C a proto se jím nebudeme dále zabývat. Podrobnější informace lze nalézt v [10].

Firma Microsoft dodává pro překlad jazyka IDL kompilátor MIDL. Ten vytváří z popisu rozhraní proxy dll pro zajištění marshallingu dat a dále zdrojové soubory pro jazyk C obsahující deklaraci všech identifikátorů, konstant a objektů.

2.1.5 Rozhraní IUnknown

COM definuje několik standardních rozhraní. Tím základním, které musí povinně každá komponenta implementovat a od kterého jsou všechny další rozhraní odvozeny, je IUnknown.

Toto rozhraní definuje následující tři funkce zajišťující základní operace nutné pro každou komponentu:

- **HRESULT AddRef()** – inkrementuje vnitřní počítadlo referencí pro dané rozhraní,
- **HRESULT Release()** – dekrementuje vnitřní počítadlo referencí pro dané rozhraní,
- **HRESULT QueryInterface(REFIID riid, void ** ppvObject)** – podle předaného identifikátoru rozhraní vrátí metoda ukazatel na toto rozhraní, pokud jej objekt implementuje.

Každý COM objekt si udržuje vnitřní počítadlo udávající počet referencí na každé rozhraní. Pokud jsou všechna počítadla na nule, je možné objekt odstranit z paměti. Metoda *QueryInterface*, pokud proběhne úspěšně a vrátí ukazatel na požadované rozhraní, automaticky inkrementuje počítadlo tohoto rozhraní. Metodu *AddRef* je tedy nutné volat pouze v případě, že vytváříme kopii ukazatele na rozhraní. Po ukončení používání rozhraní je nutné zavolat metodu *Release*, aby došlo ke správnému uvolnění paměti.

V systému Windows jsou všechny COM objekty, které nejsou po dobu pěti minut používány, odstraněny z paměti. Tomu lze zamezit umělým zvýšením počítadla referencí pomocí *AddRef*.

Metoda *QueryInterface* musí splňovat několik požadavků, které lze shrnout do následujících bodů:

- Adresa na dotázané rozhraní jednoho objektu musí být vždy stejná. To můžeme použít například pro zjištění, zda dva ukazatele na různá rozhraní ukazují na stejný objekt.
- Jestliže metoda jednou vrátí ukazatel na požadované rozhraní, musí jej vrátit kdykoliv během života objektu.
- Musí být reflexivní, tj. být schopna vrátit i ukazatel na rozhraní, jehož pomocí byl vznesen dotaz.
- Musí být tranzitivní, tj. pokud z rozhraní A získáme ukazatel na B a z B na C, musí být možné získat i z A ukazatel na C.
- Musí být symetrická, tj. umožňovat získání ukazatele z B na A pokud jsme ukazatel na B předtím získali z A.

2.1.6 Vytváření objektů a ClassObject

Pro vytváření COM objektů nemá klientská aplikace jiné informace než CLSID žádaného objektu, případně jeho textovou obdobu ProgID. Pro převod ProgID na CLSID existuje API funkce *CLSIDFromProgID*, která vyhledá příslušný záznam v registrech systému.

Pro vytvoření objektu pomocí známého CLSID pak stačí zavolat funkci *CoCreateInstance* (pro objekty vytvářené na počítači, na kterém běží klientská aplikace), nebo *CoCreateInstanceEx* (pro objekty vytvářené na vzdáleném počítači). *CoCreateInstance* je definována jako:

```
STDAPI CoCreateInstance (REFCLSID rclsid, LPUNKNOWN pUnkOuter, DWORD
dwClsContext, REFIID riid, LPVOID *ppv);
```

Kde:

- `rclsid` je identifikátor objektu, který chceme vytvořit,
- `pUnkOuter` udává, zda má být objekt součástí agregace (viz. 2.1.7) a případně ukazuje na rozhraní IUnknown rodičovského objektu,
- `dwClsContext` specifikuje požadovaný kontext, ve kterém má být spuštěn kód objektu,
- `riid` je identifikátor požadovaného rozhraní a
- `ppv` je ukazatel na místo, kam má být uložen ukazatel na požadované rozhraní.

Funkce *CoCreateInstanceEx* má syntaxi velmi podobnou. Dalším parametrem je specifikován cílový počítač, na kterém má být objekt spuštěn.

Pokud má být COM objekt vytvářen pomocí výše uvedených funkcí, je potřeba implementovat zvláštní objekt ClassObject, který zajistí správné vytváření požadované komponenty v závislosti na způsobu použití (local, remote, in-process). Tento objekt není vytvářen pomocí *CoCreateInstance*, ale pomocí zvláštní funkce *CoGetClassObject*. Ta pomocí záznamů v registrech rozliší o jaký typ zdrojového souboru jde (DLL, OCX nebo EXE) a podle toho určí postup jak ClassObject získat. Pro DLL a OCX soubory je to pomocí volání funkce *DllGetClassObject* deklarované jako:

```
STDAPI DllGetClassObject(const CLSID &rclsid, const IID &riid,
void ** ppv);
```

COM předá této funkci CLSID požadovaného objektu a v parametru `ppv` je vrácen ukazatel na příslušný ClassObject.

Pro soubory typu EXE musí server po spuštění zaregistrovat pro každou komponentu příslušný ClassObject pomocí *CoRegisterClassObject*. Zaregistrované objekty jsou uchovávány ve specielním seznamu a podle potřeby vytvářeny. Po skončení činnosti serveru jsou zaregistrované objekty odebrány ze seznamu pomocí *CoRevokeClassObject*. Spuštění EXE souboru nebo nahrání DLL či OCX knihovny zajistí COM automaticky.

Pro standardní vytváření COM objektů je vyžadováno pouze jediné rozhraní, které musí ClassObject implementovat. Je jím IClassFactory, které kromě funkcí zděděných od IUnkown, deklaruje dvě další:

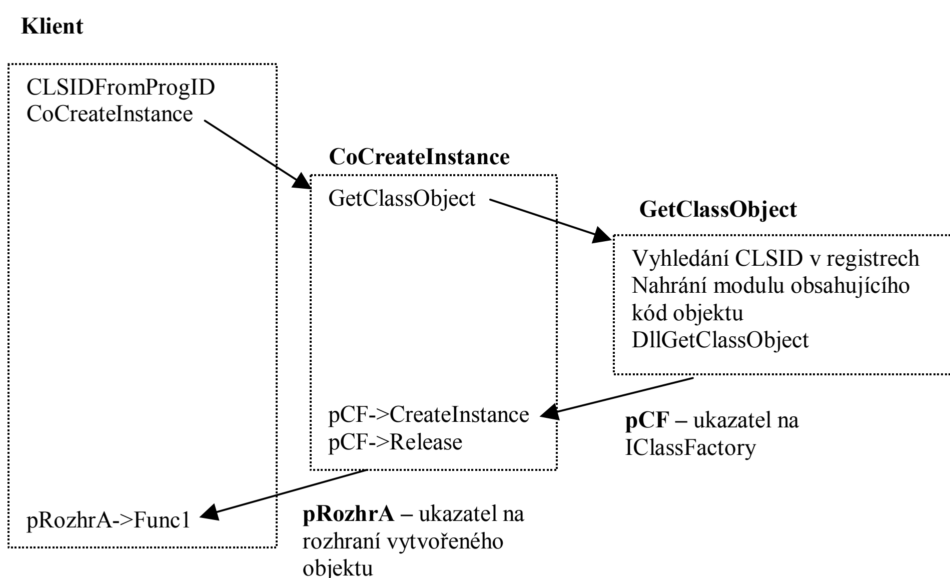
```
STDMETHODIMP CreateInstance(IUnknown *pUnkOuter, REFIID iid,
                                void ** ppv);
STDMETHODIMP LockServer(BOOL);
```

Metoda *CreateInstance* je srdcem celého ClassObjectu a zajišťuje vytváření příslušné komponenty. Protože v COM neexistuje nic takového jako ukazatel na objekt, ale pouze ukazatele na rozhraní objektů, vrací tato metoda ukazatel na rozhraní specifikované v parametru `iid`. Ten je umístěn do paměti odkazované parametrem `ppv`. Současně je automaticky inkrementován počet referencí na toto rozhraní, takže vytvořený objekt zůstane v paměti.

Metoda *LockServer* umožňuje uzamčení ClassObjectu v paměti tak, že není uvolněn ani při nulovém počtu referencí. To umožňuje urychlit vytváření dalších objektů.

Funkce *CoCreateInstance* tedy nejprve zavolá funkci *CoGetClassObject*, která vyhledá záznam příslušného CLSID v registrech a podle toho určí typ serveru a získá ukazatel na příslušný ClassObject, nebo vytvoří nový. Dále zavolá metodu *CreateInstance* rozhraní IClassInterface získaného ClassObjectu a tím obdrží ukazatel na požadované rozhraní požadovaného objektu. Nakonec uvolní ClassObject z paměti. Tento postup je znázorněn na obr. 2.3.

Takový proces může být poměrně časově náročný. Pokud je tedy už dopředu známo, že aplikace bude potřebovat vytvořit více objektů stejného typu, doporučuje se nepoužívat funkci CoCreateInstance. Místo toho je možné získat přímo ukazatel na IClassFactory a zamknout ClassObject v paměti. Při vytváření dalších objektů pak nemusí být opětovně prohledávány záznamy v registrech a operace může probíhat téměř stejně rychle jako při použití operátoru **new**.



obr. 2.3 Schéma vytváření COM objektu pomocí CoCreateInstance

2.1.7 Agregace objektů

Pro většinu složitějších COM objektů platí, že jsou samy postaveny jako COM aplikace. Tedy že ke zprostředkování svých služeb využívají služeb jiných objektů. V terminologii COM je to označováno jako agregace. Dílčí objekty jsou nazývány vnitřními objekty agregace, výsledný produkt je pak vnějším objektem. V mnoha případech vnější objekt přímo publikuje některé z rozhraní svých vnitřních objektů. Z pohledu klienta se však stále musí jednat o jedinou komponentu, pro kterou platí všechna pravidla uvedená v předchozích odstavcích.

Pokud tedy klient pomocí metody *QueryInterface* na vnějším objektu obdrží ukazatel na jedno z rozhraní vnitřního objektu, musí být také možné tento postup obrátit. Vnitřní objekty tak ale musí mít informace o tom, že jsou součástí většího celku a musí mít k dispozici ukazatel na nadřazený objekt. K tomuto účelu slouží parametr `pUnkOuter` metody *CreateInstance*. Pomocí tohoto parametru předá vnější objekt ukazatel na své rozhraní IUnknown. Vnitřní objekty pak mohou ve svých metodách *QueryInterface* přesměrovat všechny dotazy na rozhraní, které nepodporují, na nadřazený objekt.

2.1.8 Rozšíření základního modelu COM

Ze základního modelu COM vzniklo postupně několik jeho rozšíření umožňující vytvářet specializované aplikace. Tyto specifikace definují rozhraní a datové typy, které musí každý objekt vyhovující dané specifikaci implementovat.

Nyní uvedeme výčet nejdůležitějších rozšíření a jejich stručný popis.

Automation servers – aplikace podporující tuto specifikaci nabízejí možnost kontrolovat a vytvářet vlastní objekty pomocí COM rozhraní z jiných aplikací. Asi nejznámějším příkladem jsou programy balíku MS Office, které umožňují touto cestou vytvářet a upravovat své dokumenty. Všechna makra a programy jazyka Visual Basic používají právě tuto metodu komunikace. Pomocí publikovaných rozhraní je možné provádět prakticky všechny funkce dostupné pomocí klasických ovládacích prvků.

Jediným požadavkem, který musí Automation Server splňovat, je implementace rozhraní IDispatch popsaného blíže v 2.1.10.

ActiveX controls – jsou asi nejrozšířenějším použitím COM specifikací. Jsou velmi často využívány v internetových dokumentech. Jedná se o specializované in-proccess servery většinou určené pro vložení do klientských aplikací jako grafické komponenty. Specifikace definuje sadu rozhraní určených pro kontrolu zobrazování těchto komponent. Všechny vlastnosti a metody jsou publikovány pomocí Automation rozhraní (2.1.10). Většinou se nejedná o celé

funkční aplikace, ale pouze o předem vytvořené části, ze kterých je aplikace sestavena. Prohlížeče webových dokumentů podporují automatické stažení a registrování těchto komponent na klientském počítači.

Jako typický příklad je možné uvést různá tlačítka a další ovládací prvky vyskytující se v HTML dokumentech.

Active Server object – objekty určené pro použití v HTML stánkách generovaných pomocí Active Server Pages skriptů. Tyto objekty jsou, na rozdíl od ActiveX komponent, vždy spouštěny na serveru. Nemusí se vždy jednat o vizuální objekty. Jsou používány například pro připojení k databázím umístěným na serveru.

Active documents – sada rozhraní podporujících vkládání a editování datových objektů různých typů v jednom dokumentu. Příkladem mohou být dokumenty aplikací MS Word nebo MS Excel, které umožňují vkládat a editovat celou škálu dokumentů jiných aplikací jako jsou obrázky, grafy, tabulky, multimediální soubory atd.

2.1.9 Typové knihovny

Protože všechny COM objekty jsou distribuovány v binární podobě bez zdrojových kódů, není pro klientskou aplikaci možné získat z nich informace o rozhraních a datových typech, které podporují. Z toho důvodu byly zavedeny typové knihovny.

Typové knihovny jsou v naprosté většině případů vytvářeny z popisu objektu v jazyce IDL (2.1.4). Obsahují informace o všech objektech a rozhraních definovaných v popisované knihovně včetně jejich GUID. Pro každé rozhraní pak seznam metod s počty a typy parametrů. Dále obsahují dispatch identifikátory (dispIDs) pro Automation rozhraní. Rovněž mohou obsahovat:

- popisy uživatelských typů spojených s definovanými rozhraními,
- metody publikované Automation nebo ActiveX serverem, které nejsou metodami rozhraní,
- informace o výčtových typech (enumerations), záznamech (records) a aliasech,
- reference na typy deklarované v jiné typové knihovně.

Typové knihovny velmi usnadňují použití COM objektů. Jsou vyžadovány pro ActiveX a Automation objekty. Pro ActiveX objekty musí být přímo přilinkovány ke knihovně, obsahující kód samotné komponenty (DLL, OCX). Pro všechny objekty používající duální rozhraní (2.1.10) umožňují typové knihovny zrychlit provádění operací pomocí tzv. *early binding* (statické spojování s objektem).

V typové knihovně je, jak již bylo zmíněno, u každé metody uvedeno její dispID a proto je možné provést náhradu volání metod pomocí IDispatch za přímé volání metod jednotlivých

rozhraní už v době kompilace aplikace. Není tak třeba vždy znovu získávat ukazatel na volanou metodu pomocí IDispatch. Tato technologie, nazvaná early binding, může při častém volání metod rozhraní celou aplikaci výrazně urychlit.

Přístup k informacím uloženým v typových knihovnách je možný pomocí sady specializovaných rozhraní:

- `ITypeLib` – zpřístupňuje definice datových typů,
- `ITypeLib2` – rozšiřuje předchozí o podporu pro řetězce s dokumentací, uživatelská data a statistiky o typové knihovně,
- `ITypeInfo` – zpřístupňuje informace o jednotlivých objektech definovaných v typové knihovně a o jejich rozhraních,
- `ITypeInfo2` – rozšiřuje předchozí o možnosti přístupu k dalším informacím obsaženým v typové knihovně,
- `ITypeComp` – umožňuje rychlý přístup k informacím o dispID nutný pro překladače podporující early binding.

Většina moderních programovacích prostředí (jako např. C++ Builder, Visual C++, Visual Basic) však práci s těmito rozhraními vykonává automaticky.

2.1.10 IDispatch a Dual rozhraní

Všechny objekty podporující Automation nebo ActiveX technologie komunikují se svými klienty pomocí rozhraní IDispatch.

Toto rozhraní je odvozeno přímo od IUnknown a jeho deklarace v jazyce IDL vypadá následovně:

```
[object,uuid(00020400-0000-0000-C000-000000000046), pointer_default(unique)]
    interface IDispatch : IUnknown
    {
     typedef [unique] IDispatch * LPDISPATCH;
     HRESULT GetTypeInfoCount( [out] UINT * pcinfo);
     HRESULT GetTypeInfo(
            [in] UINT iTInfo,
            [in] LCID lcid,
            [out] ITypeInfo ** ppTInfo
            );
     HRESULT GetIDsOfNames(
            [in] REFIID riid,
            [in, size_is(cNames)] LPOLESTR * rgszNames,
            [in] UINT cNames,
```

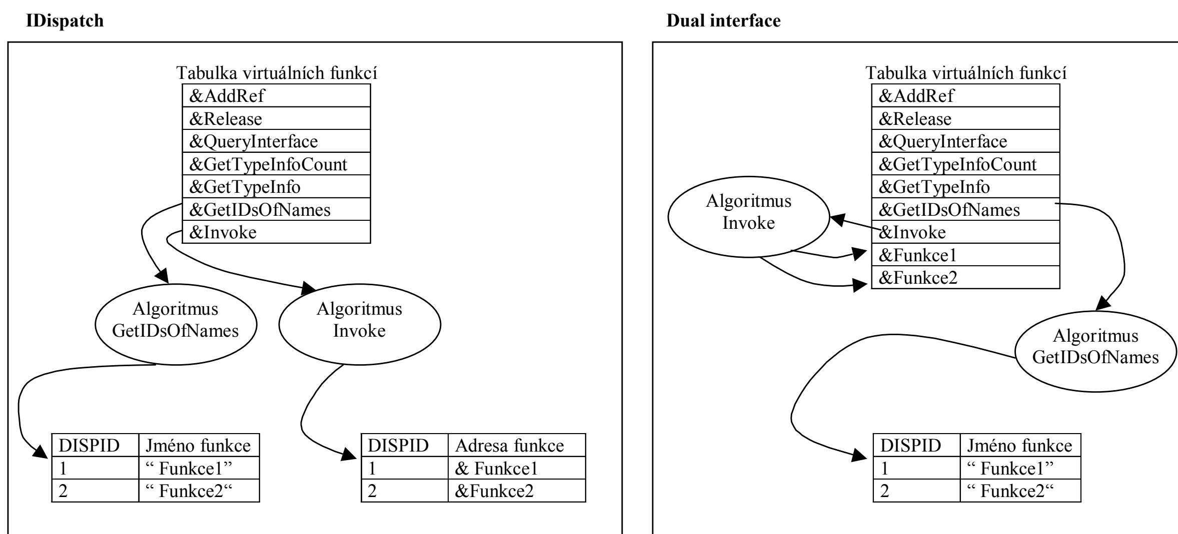
```
        [in] LCID lcid,
        [out, size_is(cNames)] DISPID * rgDispId
        );
    HRESULT Invoke(
        [in] DISPID dispIdMember,
        [in] REFIID riid,
        [in] LCID lcid,
        [in] WORD wFlags,
        [in, out] DISPPARAMS * pDispParams,
        [out] VARIANT * pVarResult,
        [out] EXCEPINFO * pExcepInfo,
        [out] UINT * puArgErr
        );
}
```

Účelem IDispatch je další usnadnění komunikace mezi klientem a COM objektem. Všechny funkce serveru jsou volány pomocí metody *Invoke* tohoto rozhraní. Jako jeden z parametrů této metody je identifikační číslo požadované funkce. Pro získání tohoto identifikátoru slouží druhá metoda rozhraní IDispatch s názvem *GetIDsOfNames*. Ta dokáže přeložit jména funkcí serveru v řetězcové podobě na příslušné identifikátory.

Předávání parametrů volaným funkcím se děje pomocí parametru pDispParams funkce *Invoke*. Všechny ale musí být tzv. Automation compatible typu. Pro tyto datové typy je zajištěn automatický marshalling dat implicitními proxy/stub knihovnami. Všechny předané parametry jsou „zabaleny“ do jednoho balíku a předány implementaci metody *Invoke* na serveru. Ta musí balík rozbalit, zavolat příslušnou metodu a vrátit zpět návratové hodnoty. Implementace *Invoke* musí také zajistit obsloužení všech chybových situací. Tímto způsobem lze nejen volat metody, ale také číst a zapisovat hodnoty všech *property* objektů.

Způsob volání metod pomocí *GetIDsOfNames* a *Invoke* je nazýván late binding (dynamické spojování s objektem). Při častém volání metod je to ale způsob neefektivní. I když je samozřejmě možné získaný identifikátor z *GetIDsOfNames* uložit a volat pouze metodu Invoke, je stále nutné podle předaného identifikátoru zjistit ukazatel na příslušnou funkci, zavolat ji a návratové parametry předat zpět klientovi.

Pro urychlení volání funkcí existují tzv. Dual rozhraní. Tato rozhraní podporují jak klasické volání funkcí rozhraní pomocí tabulky virtuálních funkcí, viz kap. 2.1.1, tak pomocí IDispatch. Pro duální rozhraní může potom překladač, který to umožňuje, použít early binding (viz 2.1.9) a proces urychlit. Rozdíl mezi IDispatch a Duálním rozhraním je na obr. 2.4.



obr. 2.4 Schéma volání funkcí pomocí IDispatch a duálního rozhraní

2.1.11 Threading model

V případě, že programátor předpokládá současné připojení několika klientů k jediné aplikaci serveru (i když pro každého klienta je samozřejmě vytvářen vlastní objekt, ale všechny jsou součástí jedné aplikace), je výhodné zabudovat do aplikace podporu pro více vláken (paralelně běžících programů v rámci jednoho procesu). V případě funkcí, jejichž vykonání je časově náročné, to může přinést značné zlepšení výkonu serveru. Znamená to ovšem také větší nároky na implementaci serveru. COM podporuje několik modelů vytváření a používání vláken v serveru. Podle zvoleného modelu je nutné, aby některé nebo všechny části serveru byly implementovány jako thread safe, jinak může docházet ke konfliktům, jejichž příčiny je velmi obtížné odhalit.

Rozeberme nyní podrobněji jednotlivé modely.

Single – je nejjednodušší model, bez podpory vláken. Všechny COM objekty serveru jsou vytvořeny v jediném (hlavním) vlákně a pouze z tohoto vlákna k nim může být přistupováno. COM zajistí, že všechna volání funkcí od klientů jsou prováděna postupně a proto odpadají jakékoliv starosti o kontrolu přístupu k lokálním i globálním proměnným. Zároveň ale, pokud jeden z klientů zavolá nějakou z funkcí serveru, musejí ostatní klienti počkat na její dokončení, dokud nebudou obslouženy jejich požadavky. Proto je tento model vhodný pouze pro jednoduché servery, kde se nepředpokládá přístup mnoha klientů a kde čas potřebný pro provedení funkcí rozhraní je relativně krátký.

Apartment – každý z COM objektů je vytvořen ve svém vlastním vlákně a pouze z tohoto vlákna mohou být volány jeho metody. Při použití tohoto modelu mohou objekty bezpečně

přistupovat ke svým lokálním datům. Všechny zápisy do globálních dat aplikace ale musí být zajištěny proti konfliktům vznikajícím z přístupu z více vláken najednou (např. pomocí kritických sekcí). Lokální data jsou bezpečná i při volání více metod za sebou, protože metody mohou být volány pouze jedním klientským vláknem. Použití tohoto modelu již přináší značné zlepšení výkonu serveru.

Free – každý vytvořený objekt může být volán libovolným počtem vláken v libovolném čase. Objekty musí ochránit jak globální tak i lokální data instance proti konfliktům způsobeným přístupem z více vláken. Lokální data jednotlivých metod jsou vždy bezpečná, protože jsou uložena v zásobníku, který má každé vlákno vlastní. Tento model přináší maximální zlepšení výkonu serveru. Jeho použití je zvláště vhodné pro vzdálené servery, kde je vhodnější než model Apartment. Při volání metody na vzdáleném serveru je totiž toto volání přijato jedním vláknem z tzv. thread pool (zásobárny vláken). Veškeré předávané parametry metody jsou pomocí marshallingu přemístěny do adresového prostoru tohoto vlákna. V případě použití modelu Free, pak může vlákno serveru přímo zavolat metodu objektu. Pro model Apartment je ale nutné přemístit toto volání do vlákna příslušného objektu, což samozřejmě vyžaduje čas navíc, neboť je nutné opět provést marshalling, i když jen lokálně.

Všechny ActiveX komponenty typicky používají Apartment model.

2.2 Struktura COM aplikace

2.2.1 Alokace paměti pro přenášené parametry

Jedním z nejdůležitějších pravidel, kterými je nutné se při implementaci COM aplikací řídit, jsou pravidla o práci s pamětí. Protože veškeré předávané parametry musí být vždy nejprve přemístěny do adresového prostoru serveru a poté pak navrácené parametry zpět ke klientovi, je potřeba se při práci s pamětí držet několika pravidel. Prvním a nejdůležitějším pravidlem je, že klient je vždy zodpovědný za uvolnění veškeré paměti spojené s parametry a to i s parametry typu [out] nebo [in/out]. Při nedodržení tohoto pravidla dochází k „únikům“ paměti.

Další pravidla se týkají alokace paměti. Jejich přehledné shrnutí pro různé typy parametrů je v následující tabulce. Typ parametrů je specifikován pomocí jazyka IDL popsaného blíže v 2.1.4.

tabulka 2-A Typy parametrů funkcí rozhraní

Popis parametru	Význam pro správu paměti
[in] typ*	Klient předá adresu parametru daného typu, pro který alokoval paměť jakýmkoliv způsobem. Server může tento parametr číst, ale ne měnit.
[out] typ*	Klient předá adresu parametru daného typu, pro který alokoval paměť jakýmkoliv způsobem. Server nemůže tento parametr číst, ale pouze do něj zapisovat.
[in,out] typ*	Klient předá adresu parametru daného typ, pro který alokoval paměť jakýmkoliv způsobem. Server může tento parametr číst i měnit.
[out] typ**	Klient předá adresu ukazatele na daný typ, pro který alokoval paměť jakýmkoliv způsobem. Server použije IMalloc pro alokaci nového parametru daného typu (případně pole) a adresu této paměti uloží do ukazatele.
[in,out] typ**	Klient alokuje paměť pro parametr daného typu (nebo pole) pomocí IMalloc a předá serveru ukazatel na tuto paměť. Server tuto paměť uvolní a pomocí *CoTaskMemAlloc* alokuje jinou, do které uloží návratové hodnoty. Ukazatel pak předá zpět klientovi.

Z tabulky je patrné, že pokud je předávaným parametrem pole, jehož velikost není předem známa a které tudíž musí být předáno pomocí ukazatele na ukazatel na první prvek tohoto pole, je nutné aby samotné pole bylo alokováno pomocí funkcí, které poskytuje rozhraní IMalloc. Jenom tak je možné zajistit přenos i této části paměti do cílového adresového prostoru a správné předání celého pole.

Ukazatel na rozhraní IMalloc je možné získat pomocí API funkce *GetMalloc*. To je ale většinou nutné pouze, pokud implementujeme vlastní mechanismus zajišťující marshalling. Pro pouhé alokování a uvolňování paměti stačí použít API funkce *CoTaskMemAlloc* a *CoTaskMemFree*, které mají stejnou syntaxi jako jejich standardní protějšky malloc a free a které používají pro práci s pamětí ukazatel na IMalloc získaný automaticky při inicializaci rozhraní COM.

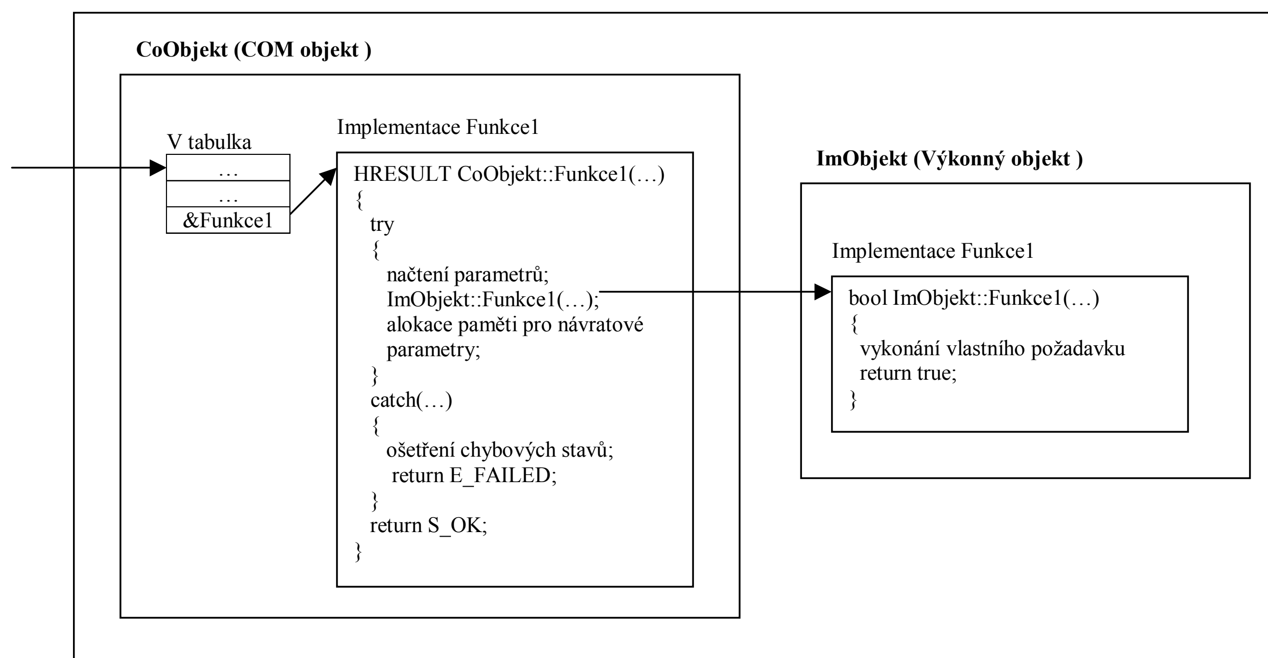
2.2.2 COM Server

Základem pro implementaci všech metod rozhraní je ošetření veškerých chybových stavů, které mohou při běhu metody nastat. Je pravidlem, že musí být v rámci metody ošetřeny všechny výjimky, které mohou nastat. Prakticky všechny metody rozhraní mají jako návratovou hodnotu

typ HRESULT. Doporučuje se tedy, aby veškerý kód metody byl „obalen“ příslušným blokem **try – catch** a v druhé části tohoto bloku se zachycené výjimky převáděly na příslušný chybový kód.

Ve většině případů platí, že COM objekty ze svých metod volají metody jiných objektů aplikace, které zajišťují samotné vykonávání požadavků. Tato struktura je naznačena na obr. 2.5. Metody COM objektů pak zajišťují především ošetření chybových stavů a správnou práci s pamětí. Pokud ovšem více COM objektů používá stejný výkonný objekt, musí být volání metod výkonného objektu zajištěno proti kolizím, například pomocí kritických sekcí.

Je třeba také klást důraz na to, aby byly výstupní parametry z metod rozhraní vždy definovány. V případě, že metoda skončí s chybou (např. E_OUTOFMEMORY), je nutné, aby všechny návratové ukazatele byly nastaveny na NULL. Jedině tak lze zabránit kolizím při uvolňování paměti.



obr. 2.5 Ukázka struktury aplikace pracující jako COM server

2.2.3 Connection Point

COM ve své podstatě umožňuje pouze jeden způsob komunikace mezi klientem a serverem - volání metod publikovaných COM serverem. To však v mnoha případech nestačí. Velmi často se ukazuje potřeba komunikace na bázi událostí – tedy iniciovaná serverem. Proto byly vytvořeny tzv. přípojné body, které volání událostí umožňují.

Při použití přípojných bodů se vlastně otáčí role. Ze serveru se stává klient, který volá metody rozhraní publikované klientem a tím podává zprávy o vzniklých událostech. Přípojné

body jsou zvláštní rozhraní, na kterých mohou klienti zaregistrovat ukazatele na jimi publikované rozhraní, na kterém si přejí dostávat zprávy o nastalých událostech. Na klientem registrovaném rozhraní musí být implementovány metody pro všechny události, které mohou být serverem na daném rozhraní generovány.

Pro implementaci přípojných bodů je potřeba na straně klienta implementovat všechna rozhraní, na kterých klient požaduje přijímat události. Definice těchto rozhraní je uvedena v typové knihovně serveru, kde jsou rovněž definovány všechny povinné metody pro tato rozhraní. Na straně serveru je pak nutné implementovat podporu pro sadu rozhraní definovaných standardem COM. Uveďme nyní jejich stručný popis.

IConnectionPointContainer umožňuje klientům získat přehled o podporovaných přípojných bodech a o již navázaných spojeních. Má definovány pouze následující tři metody:

- **EnumConnectionPoints -** tato metoda vrací ukazatel na rozhraní IEnumConnectionPoints, což je standardní výčtové rozhraní odvozené od IEnum, umožňující získat seznam všech podporovaných přípojných bodů a ukazatelů na příslušné IConnectionPoint (viz níže).
- **FindConnectionPoint** dokáže na základě poskytnutého identifikátoru vrátit ukazatel na příslušný přípojný bod reprezentovaný rozhraním IConnectionPoint (viz níže).
- **EnumConnections** je další metoda, která vrací ukazatel na výčtové rozhraní (IEnumConnections), tentokrát však s výčtem všech spojení na přípojném bodě podle zadaného identifikátoru.

IConnectionPoint je rozhraní, které reprezentuje samotný přípojný bod. Nejdůležitější metodou tohoto rozhraní je metoda ***Advise***, pomocí níž klient registruje své rozhraní na serveru. Navázání spojení pro příjem událostí se děje podle následujícího schématu:

1. Klient získá pomocí *QueryInterface* ukazatel na rozhraní IConnectionPointContainer implementované serverem.
2. S použitím metod *EnumConnectionPoints* nebo *FindConnectionPoint* získá klient ukazatel na příslušné rozhraní IConnectionPoint, jehož události chce připojit.
3. Klient zavolá metodu *Advise* rozhraní IConnectionPoint, které jako parametr předá ukazatel na své rozhraní IUnknown.
4. Server pomocí *QueryInterface* na obdrženém IUnknown získá ukazatel na rozhraní, které obsahuje koncové metody (sinks) a ty pak budou volány při vzniku jednotlivých událostí.

Je tedy patrné, že klient musí implementovat jak rozhraní IUnknown, tak i rozhraní na kterém budou přijímány události (může se jednat o totéž rozhraní).

3 Standardy OPC

Technologie OPC (OLE for Process Control) je založena na rozhraní COM popsaném v kapitole 2. OPC je skupina specifikací definujících COM objekty a jejich rozhraní, použitelné pro různé účely v řídicí technice. Tyto specifikace jsou vydávány a udržovány organizací OPC Foundation [12], která sdružuje přední světové výrobce hardwaru a softwaru pro řídicí techniku.

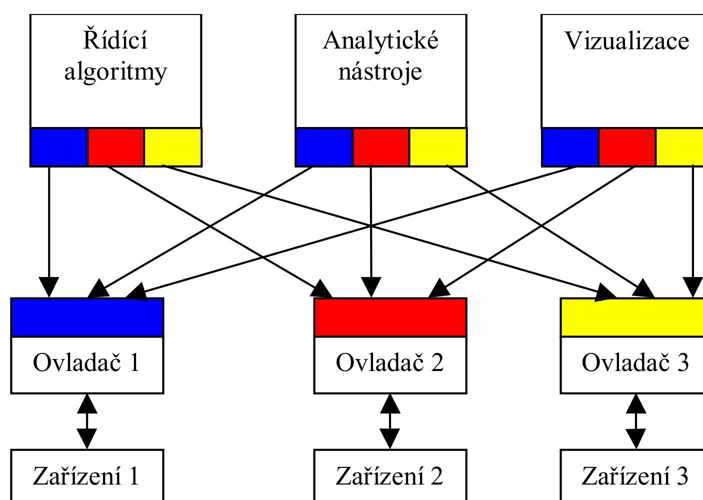
V této kapitole se nejprve budeme zabývat obecnými informacemi o OPC specifikacích, jejich druhy a popisem obecné architektury OPC aplikací. V dalších částech se budeme podrobněji zabývat standardy OPC Data Access a OPC Historical Data Access, které byly použity pro vypracování této diplomové práce.

Většina informací uvedených v následujících odstavcích je převzata z materiálů dostupných na [12].

3.1 Obecné informace

3.1.1 Důvod vzniku

Důvodem vzniku OPC byla snaha sjednotit výměnu dat mezi zařízeními používanými v řídicí technice, a to i pro zařízení od různých výrobců. Existuje mnoho aplikací vyžadujících přístup k datům z řídicích systémů. Ať už se jedná o „real time“ data používaná jak pro vizualizaci, tak pro další řízení a rozhodování nebo o data „historická“, uchovávaná v databázích pro použití k analýze a optimalizaci řídicích procesů.

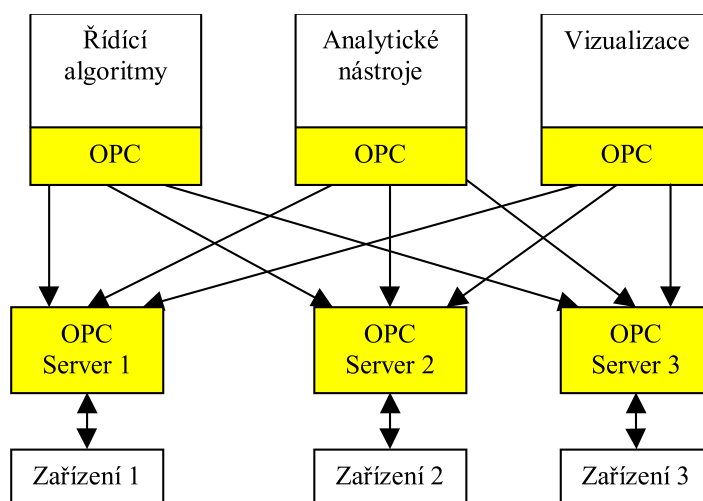


obr. 3.1 Přístup k zařízením s použitím specifických ovladačů (Převzato z [3])

Mnoho z těchto aplikací používá pro přístup k datům specifických ovladačů odlišných pro každé zařízení, případně databázi. To přináší mnoho problémů:

- Je potřeba vyvíjet nové ovladače pro každé připojitelné zařízení.
- Ovladače dodávané od různých výrobců nepodporují vždy stejné funkce.
- Změna v hardwarové konfiguraci může způsobit nefunkčnost některých ovladačů.
- Dvě různé aplikace většinou nemohou přistupovat k jednomu zařízení najednou, protože používají různé ovladače.

Výrobci zařízení se snaží řešit tyto problémy dodáváním univerzálních ovladačů pro svoje zařízení, ale jejich snaha je bržděna různými požadavky klientů. Je téměř nemožné vyvinout efektivní ovladač, který by uspokojil požadavky všech klientů.



obr. 3.2 Přístup k zařízením s použitím OPC (Převzato z [3] a upraveno)

OPC vytváří pomyslnou hranici mezi výrobci hardwarových zařízení a vývojem uživatelských aplikací. Dává k dispozici univerzální mechanismy pro standardizovaný přenos dat mezi serverem a klientem. S použitím OPC může výrobce hardwaru vyvinout pro své zařízení efektivní server komunikující se zdrojem dat který, je použitelný pro všechny klienty.

V dnešní době je již pro většinu renomovaných výrobců hardwaru samozřejmostí dodávat se svými zařízeními OPC Servery umožňující přístup k datům, která zařízení poskytuje.

Rozdíl mezi přístupem k zařízením pomocí jednotlivých ovladačů a pomocí OPC je nejlépe patrný z obr. 3.1 a obr. 3.2.

3.1.2 Typy OPC specifikací

V důsledku potřeby přístupu k různým typům dat bylo nutno vytvořit několik standardů. Vznikla tak celá rodina specifikací, která se i nadále rozrůstá, aby uspokojila všechny potřeby moderní výměny dat. Uvedeme zde pouze přehled dosud publikovaných nebo připravovaných specifikací. V následujících kapitolách se pak budeme podrobněji zabývat dvěma z tohoto přehledu.

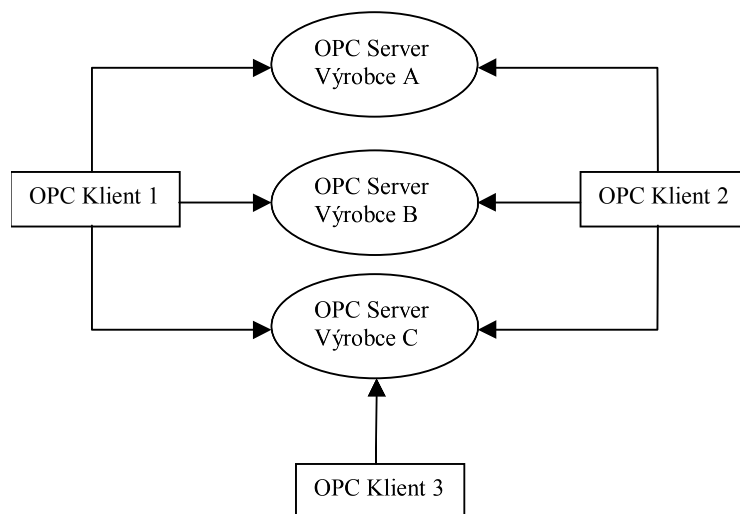
- **Data Access** - slouží pro přístup k datům v reálném čase. Používá se např. pro rozhodovací a řídicí algoritmy, vizualizaci a pro zaznamenávání dat do databází. Podrobněji se jím zabýváme v 3.2.
- **Alarms and Events** – pomocí tohoto rozhraní může klient získat informace o vzniku událostí v systému. Jsou definovány dva základní typy událostí. **Alarm** je definován jako abnormální stav buď samotného serveru, nebo jedné z položek, která je serverem sledována. Je tedy vždy spojen s nějakým stavem. Například pokud hodnota sledované položky překročí stanovený limit, je vyvolán alarm HighLimit. **Event** je událost spojená s výskytem určitého jevu. Například zásah operátora nebo výskyt chyby v systému jsou typické ukázky takových událostí. Klient má možnost prohlížet všechny definované události, zjišťovat jejich stav a být informován o vzniku událostí.
- **Historical Data Access** – rozhraní sloužící pro získávání dat z databází, kde jsou zaznamenány historické vývoje sledovaných položek. Používá se zejména pro analýzu a optimalizaci řídicích systémů. Podrobněji se jím zabýváme v kapitole 3.3.
- **XML Data Access –** předpis definující jednotná, ale flexibilní pravidla a formáty pro publikování dat pomocí XML (eXtended Markup Language). Jedná se především o data zpřístupňovaná pomocí předchozích tří specifikací.
- **Batch –** specifikace pro výměnu informací s průmyslovými zařízeními pro dávkové zpracování (založena na modelu ANSI/ISA-S88.01-1995). Jsou definována rozhraní pro výměnu čtyř typů informací: možnosti zařízení, současné pracovní podmínky, historická data a receptury.
- **Data eXchange –** standard specifikující protokoly a struktury pro výměnu dat mezi různými OPC servery, mezi hardwarovým zařízením a OPC serverem a mezi dvěma zařízeními. S použitím tohoto standardu lze jednoduše integrovat do jedné sítě několik zařízení od různých výrobců.
- **Security –** pomocí rozhraní definovaných v této specifikaci lze zvýšit bezpečnost zajištění přístupu ke všem serverům vytvořeným podle předchozích standardů. Použití OPC serverů poskytujících důležité informace o řízených procesech se velmi rychle rozšiřuje. Proto i tento standard, založený na bezpečnostním modelu z Windows NT, nabývá na důležitosti.

Každá z výše uvedených specifikací definuje vlastní sadu COM objektů a rozhraní tak, aby zajistila potřebnou funkčnost. Existují však také dvě rozhraní společná a povinná pro všechny specifikace, ta jsou blíže popsána v odstavcích 3.1.5 a 3.1.6.

3.1.3 Architektura OPC aplikací

Protože specifikace OPC jsou založeny na rozhraní COM, je i architektura OPC aplikací shodná s architekturou většiny COM aplikací popsanou v 2.2.

OPC klient je typická ukázka COM klienta, který s využitím objektů serveru může získávat data z různých zařízení, nezávisle na výrobci zařízení a způsobu jeho komunikace s počítačem. Jeden klient se může připojit na libovolný počet serverů různých výrobců a tak získat najednou přístup k několika různým zařízením. Pokud klient přistupuje najednou na několik serverů, je výhodné, aby každé z těchto připojení mělo k dispozici vlastní proces. Tím je možné dosáhnout vyššího výkonu.



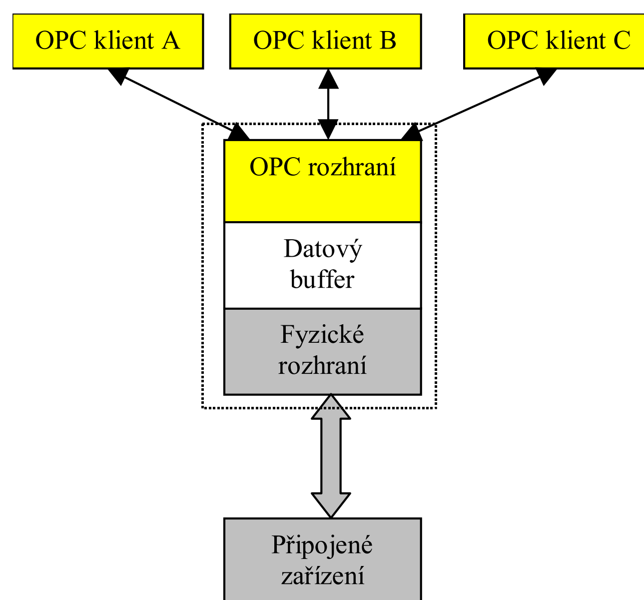
obr. 3.3 Vztah OPC Server / Klient (převzato z [5] a upraveno)

OPC specifikace předpisují pouze, jakým způsobem musí server komunikovat s klienty, ale neříkají nic o tom, jak má být tato komunikace implementována, ani jakým způsobem má server získávat data ze zařízení. Z toho důvodu se architektura serverů od různých výrobců může lišit. Obecně by mělo pro servery pracující s daty v reálném čase platit, že komunikace se zařízením a komunikace s klienty by měla být oddělena (viz obr. 3.4). Předpokládá se totiž, že komunikace se zařízením je v porovnání ke komunikaci s klientem poměrně pomalá.

Pro každý objekt mohou být ve specifikacích definována jak rozhraní povinná, tak volitelná. Povinná rozhraní slouží k zajištění základní funkčnosti objektu a jejich implementace je povinná pro všechny objekty vyhovující dané specifikaci bez výjimky. Volitelná rozhraní jsou naproti tomu určena pro zajištění dalších služeb a je na programátorovi, zda je server bude podporovat nebo ne.

Stejně jako existují volitelná rozhraní, mohou na některých rozhraních existovat i volitelné metody. V případě, že server nepodporuje volitelnou metodu, musí tato metoda vždy vracet chybový kód E_NOTIMPL.

Všichni OPC klienti by měli být napsáni takovým způsobem, aby jejich základní funkčnost nebyla závislá na nepovinných rozhraních a metodách.



obr. 3.4 Struktura OPC DA serveru

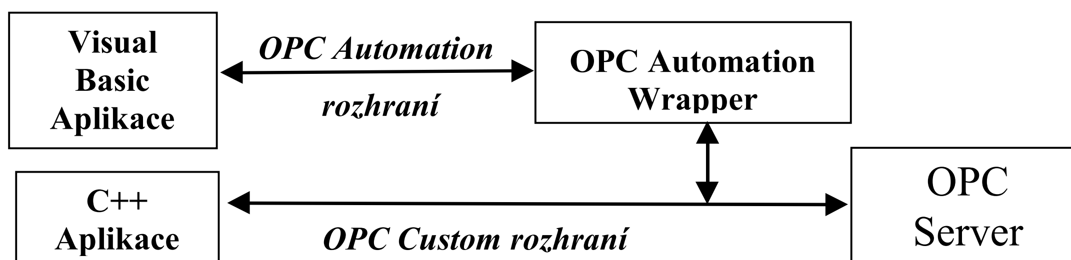
V současné době existuje již řada RAD (Rapid Application Development) nástrojů umožňujících velmi rychlé a efektivní vytváření OPC serverů. Tyto nástroje většinou zajistí implementaci té části serveru, která se týká OPC, na programátorovi je pak pouze dodání rutin pro komunikaci se zařízením.

3.1.4 Custom a Automation rozhraní

Každá z OPC specifikací obsahuje dvě sady rozhraní zajišťujících podobnou funkčnost, a to custom a automation rozhraní. Custom rozhraní je implementováno jako sada standardních COM rozhraní. Jeho účelem je použití v nižších programovacích jazycích (C, C++). Ačkoliv jeho použití je složitější než u automation rozhraní, jeho hlavní předností je vysoká efektivnost. Proto najde využití především v aplikacích náročných na rychlost komunikace (řídicí aplikace).

Automation rozhraní je pak nadstavba nad custom rozhraním, umožňující jeho funkci jako automation server. To je výhodné pro použití ve všech jazycích vyšších úrovní (Java, Visual Basic). I když není tak výkonné jako custom rozhraní, pro mnoho aplikací postačí a jeho předností je velmi snadná použitelnost. Pomocí automation rozhraní a jednoduchých skriptů ve

Visual Basicu je pak například možné přenést data ze zařízení přímo do aplikace MS Excel a tam je dále zpracovávat.



obr. 3.5 Ukázka typické OPC architektury (převzato z [5] a upraveno)

Pro své členy poskytuje OPC Foundation tzv. „wrapper dll“, které dokáží „obalit“ každé custom rozhraní a tak zajistit jeho převod na automation (viz obr. 3.5). Bylo by samozřejmě možné vytvořit rozhraní duální (2.1.10), ale ukazuje se, že není možné navrhnout ho tak, aby bylo zároveň efektivní a přitom splňovalo všechny požadavky, které uživatel očekává od automation serveru.

Každý OPC server musí povinně implementovat custom rozhraní pro danou specifikaci a volitelně může implementovat i rozhraní automation (to lze zajistit téměř automaticky pomocí „wrapper dll“).

3.1.5 Rozhraní IOPCCommon

Toto rozhraní je společné pro všechny servery vytvořené v souladu s jakoukoliv OPC specifikací. Jeho účelem je především umožnit klientům sdělit serveru kód své lokalizace (LocalID) a nastavit ho jako výchozí pro dané spojení. Toto nastavení je specifické pro každého klienta a umožňuje tedy (pokud to server podporuje), aby každý klient dostával od serveru data formátovaná podle příslušných lokálních nastavení. K tomuto účelu jsou k dispozici tři metody.

Další z metod tohoto rozhraní umožňuje klientům získat textový překlad návratových chybových kódů z jakékoliv metody všech rozhraní.

Podrobnější popis rozhraní a zmíněných metod je možné nalézt v [6].

3.1.6 Rozhraní IOPCShutdown

Účelem tohoto rozhraní je umožnit OPC Serveru vyžádat si odpojení všech klientů v případě, že má dojít k vypnutí serveru (například v důsledku zásahu uživatele). Každý klient by měl tedy implementovat toto rozhraní a zaregistrovat ho na příslušném přípojném bodě serveru tak, jak je popsáno v 2.2.3.

Na rozhraní je definována pouze jedna metoda **ShutdownRequest**. Jejím parametrem je řetězec, ve kterém může server při jejím volání sdělit klientovi důvod vypnutí serveru. Podrobnější popis rozhraní je opět uveden v [6].

3.1.7 OPC Server Browser

Pro uživatelsky příjemné použití rozhraní OPC je nutné, aby aplikace klienta mohla svému uživateli poskytnout informace o tom, jaké OPC servery jsou instalovány na jakém počítači a umožnila se k nim připojit.

Protože všechny OPC servery jsou COM komponenty, musí mít záznam v registrační databázi systému. Navíc podle specifikací OPC se musí všechny servery registrovat do systému pomocí Component Categories. To znamená, že ke svému záznamu v registrech přidají informaci o tom, jakou kategorii služeb implementují. Tyto kategorie jsou, jako všechno v COM, označeny pomocí unikátního identifikátoru, v tomto případě nazývaného CATID – Category Identifier. OPC Foundation specifikuje příslušné CATID pro každý ze svých standardů.

Ve všech operačních systémech Windows je pak instalována DLL knihovna obsahující StdComponentCatagoriesMgr (Standard Component Categories Manager). Tento COM objekt umožňuje vyhledávat v registrech všechny programy registrované pro příslušnou kategorii. Každý klient by tedy mohl využít jeho služeb k získání seznamu instalovaných OPC serverů. Problém spočívá v tom, že tato komponenta dokáže pracovat pouze in-process a je tudíž použitelná pouze pro prohlížení lokálních záznamů v registrech.

Z tohoto důvodu vytvořila OPC Foundation zvláštní program Server Browser, který může být instalován na jakémkoliv počítači a dokáže s použitím COM zprostředkovat komunikaci s lokálním StdComponentCatagoriesMgr. Doporučuje se tedy společně s instalací serveru na daný počítač nainstalovat i Server Browser (umístěn v souboru OPCENUM.EXE), jehož služeb mohou potom využívat vzdálení klienti.

3.2 Data Access 2.0

3.2.1 Účel specifikace

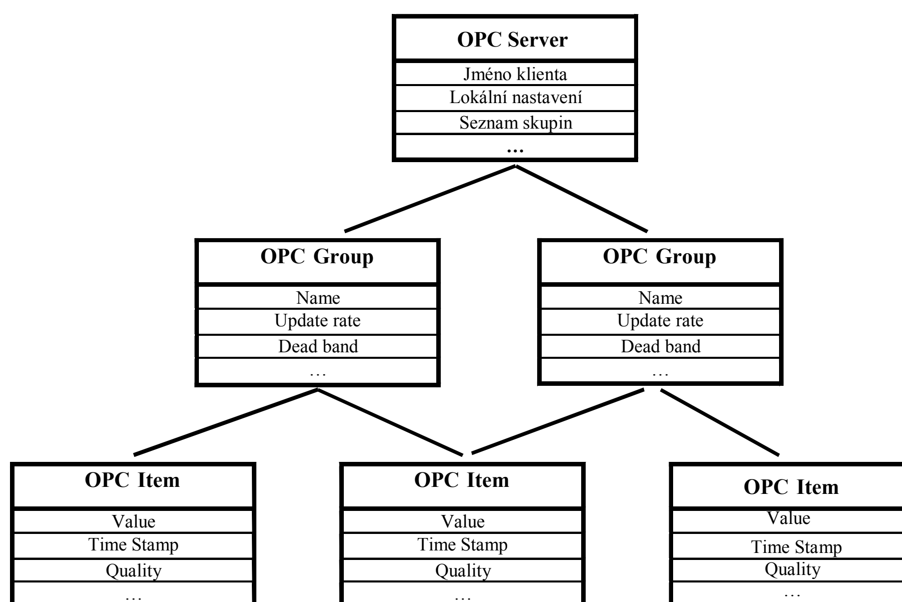
Jedná se o první z OPC specifikací (původně zamýšlena také jako jediná). Jejím účelem je umožnit aplikacím přístup k aktuálním datům z řídicích systémů. Specifikace obsahuje podporu jak pro synchronní, tak i asynchronní zápis a čtení dat. Jedná se asi o nejrozšířenější typ použití OPC standardů.

Uplatnění nachází především ve vizualizaci řídicích procesů. Protože je dnes již téměř samozřejmostí, že každý výrobce dodává ke svému řídicímu systému také OPC DA server, je možné pro vizualizaci procesu použít jakýkoliv vizualizační program podporující získávání dat z OPC.

Další významné použití OPC DA je pro ukládání dat do databází procesních dat. S použitím OPC je možné snadno získávat data z různých zdrojů a soustředit je do jediné databáze. Pro analýzu shromážděných dat lze použít další z OPC specifikací popsanou v kapitole 3.3.

3.2.2 Přehled objektů

Přístup k datům v rámci OPC DA je, jak je patrné z obr. 3.6, rozdělen do tří základních úrovní. Každá úroveň je reprezentována zvláštním objektem. Zabývejme se nyní blíže popisem jednotlivých objektů.



obr. 3.6 Struktura objektů definovaných v OPC DA

OPC Item neboli položka reprezentuje nejnižší úroveň. Představuje jednotlivé měřitelné údaje v systému. Typickým příkladem položky je například hodnota výstupu senzoru, hodnota vnitřní veličiny systému, aktuální čas na serveru apod. Základním údajem každé položky je samozřejmě její hodnota. Ta je udávána ve formě trojice: hodnota, časová značka a kvalita.

- **Hodnota** každé položky je předávána jako typ Variant, může tedy reprezentovat libovolný podporovaný datový typ.

- **Časová značka** je udávána pomocí datového typu FILETIME, který je nejkompaktnější Windows API datovou strukturou pro ukládání času. Jedná se o 64 bitový celočíselný typ reprezentující počet stovek nanosekund které uplynuly od počátku vnitřního kalendáře.
- **Kvalita** udává spolehlivost navráceného údaje. Je reprezentována osmibitovým integerem. První dva bity rozlišují tři základní stavy: DOBRÁ, ŠPATNÁ a NEZNÁMÁ. Zbývajících šest bitů může být použito pro detailnější informace.

Kromě výše zmíněné trojice je pro každou položku definována ještě celá sada dalších atributů. Ty udávají například její jméno a identifikátor, který danou položku jednoznačně určuje, datový typ hodnot, jednotky ve kterých je položka vyjádřena atd. Podrobnější popis je možné nalézt v [4].

Položky nejsou pro klienta samostatně přístupné, a proto pro ně nejsou definována žádná COM rozhraní. Přístup k nim je možný pouze prostřednictvím skupin.

OPC Group, neboli skupina, je další úroveň datového modelu OPC DA. Jejím účelem je sdružovat v sobě položky, které k sobě mají nějakým způsobem vztah. Například všechny hodnoty zobrazované na jednom displeji operátora mohou být získávány z jedné skupiny. Pomocí rozhraní definovaných pro tento objekt (viz 3.2.4) je možné manipulovat s položkami ve skupině. Server poskytuje klientům možnost automatického zasílání nových hodnot položek.

Každá skupina také obsahuje některé informace společné pro všechny její položky. Uveďme alespoň výčet těch nejdůležitějších:

- **Name** – jméno skupiny specifikované klientem. Jméno musí být unikátní v rámci spojení Server – Klient.
- **Update rate** – minimální časový interval, po kterém budou klientovi zaslány nové hodnoty položek, pokud došlo k jejich změně. Server by měl v rámci možností co nejpřesněji dodržet tento interval a testovat po jeho uplynutí podmínky pro změnu hodnoty položek.
- **Dead Band** – udává o kolik procent z rozsahu se musí minimálně změnit hodnota položky, aby byl klient informován o této nové hodnotě.

Existují dva základní typy skupin: soukromé (private) a veřejné (public). Soukromé skupiny jsou vždy vytvářeny klientem v rámci připojení a má k nim přístup pouze tento klient. Po odpojení klienta skupiny zanikají. Veřejné skupiny mohou být vytvářeny kterýmkoliv klientem nebo i serverem. Jsou pak přístupné pro všechny klienty a nezanikají ani po odpojení všech klientů.

OPC Server je nejvyšším objektem v hierarchii. Udržuje v sobě informaci o samotném spojení a slouží jako kontejner pro objekty nižší úrovně. Pomocí tohoto objektu může klient vytvářet nové objekty OPC Group, nebo se připojovat k již existujícím veřejným skupinám. Dále mu tento objekt umožňuje prohlížet si adresový prostor serveru a získávat tak informace o všech dostupných položkách. Bližší popis rozhraní je uveden v odstavci 3.2.3.

3.2.3 Rozhraní objektu OPC Server

Podle specifikace OPC Data Access musí tento objekt implementovat čtyři povinná rozhraní a může rovněž implementovat další tři volitelná rozhraní pro rozšíření svých služeb. Pro tento i všechny následující objekty se omezíme pouze na výčet definovaných rozhraní se stručným popisem jejich účelu a funkce. Podrobnější popis všech předepsaných metod a jejich parametrů je možné nalézt v [4].

- **IOPCCommon –** (povinné rozhraní) viz 3.1.5.
- **IOPCServer** – (povinné rozhraní) hlavní rozhraní serveru. Slouží pro vytváření a spravování skupin v rámci spojení klient – server.
- **IConnectionPointContainer** - (povinné rozhraní) poskytuje přístup k přípojnému bodu pro IOPCShutdown. Podrobnější popis je uveden v 2.2.3.
- **IOPCItemProperties** – (povinné rozhraní) umožňuje klientům procházet parametry spojené s každou položkou a získávat jejich hodnotu. Kromě jména, hodnoty, časové značky a kvality existuje ještě celá řada dalších parametrů definovaných specifikací a ta může být ještě rozšířena o parametry definované pouze pro daný server. Toto rozhraní není určeno pro získávání většího objemu dat. Typický příklad použití je následující: klient získá pomocí IOPCBrowseServerAddressSpace jméno požadované položky. Z rozhraní IOPCItemProperties pak obdrží seznam parametrů definovaných pro danou položku a ten nabídne uživateli. Uživatel si vybere ty, které ho zajímají (např. datový typ, fyzikální jednotky, horní a dolní limit …) a obdrží jejich hodnoty. Pro další čtení hodnot dané položky pak použije jedno z rozhraní definovaných na objektu OPC Group.
- **IOPCServerPublicGroups** – (volitelné rozhraní) pomocí tohoto rozhraní může klient získávat ukazatel na veřejné objekty OPC Group, definované buď jiným klientem nebo serverem. Je také možné tyto objekty odstranit.
- **IOPCBrowseServerAddressSpace** – (volitelné rozhraní) umožňuje klientům procházet položky definované v adresovém prostoru serveru a dát tak uživateli možnost výběru jaké položky sledovat. Adresový prostor serveru může být buď

plochý, v tom případě jsou všechny položky definovány na stejné úrovni, nebo hierarchický, pak jsou položky definovány ve stromové struktuře. Klient může získávat seznam jmen všech položek (listů) a uzlů (větví) na dané úrovni a přecházet mezi úrovněmi. Je možné také nadefinovat kritéria, podle kterých mohou být předávané seznamy filtrovány. Podpora filtrů je nepovinná.

- **IPersistFile** – (volitelné rozhraní) jedná se o implementaci standardního COM rozhraní. To umožňuje klientům dát serveru pokyn k uložení, nebo nahrání jeho aktuální konfigurace. Jedná se ovšem pouze o konfiguraci samotného serveru, nikoliv jednotlivých klientských připojení. Nejsou proto ukládány žádné informace týkající se definic položek a skupin. Konfigurací serveru se rozumí například adresy PLC, ze kterých jsou získávána data, jména databází atp.

3.2.4 Rozhraní objektu OPC Group

Tento objekt musí implementovat, kromě IUnknown osm rozhraní, z toho je jedno volitelné. Omezíme se opět na jejich pouhý výčet s uvedením základní funkce a účelu. Bližší informace o objektu jako takovém je možné nalézt v odstavci 3.2.2. Podrobnější popis metod rozhraní je pak uveden v [4].

- **IOPCItemMgt** – (povinné rozhraní) umožňuje klientům spravovat položky obsažené ve skupině. Klient může položky jak přidávat a odebírat, tak i nastavovat některé jejich chování. Je například možné nastavit, zda jsou vybrané položky v rámci skupiny aktivní nebo jaký je požadovaný typ dat pro danou položku. Je také samozřejmě možné získat seznam všech položek ve skupině.
- **IOPCGroupStateMgt** – (povinné rozhraní) slouží pro nastavování vlastností skupiny, jako jsou například update rate, dead band, jméno skupiny atd. Dále je možné vytvářet kopie skupiny, čímž je rovněž vytvářen i nový objekt.
- **IOPCPublicGroupStateMgt** - (volitelné rozhraní) je určeno pro vytváření veřejných skupin ze skupin soukromých. Každá skupina je vytvářena klientem jako soukromá. Pomocí metody tohoto rozhraní je možné z ní vytvořit skupinu veřejnou, pro kterou platí poněkud odlišná pravidla, jak bylo popsáno v 3.2.2.
- **IOPCSyncIO** – (povinné rozhraní) umožňuje klientům provádět synchronní čtení a zápis hodnot položek definovaných v rámci skupiny. Synchronní v tomto případě znamená, že operace je vždy dokončena v rámci volání dané metody. Pro čtení je možné požadovat jak čtení z cache serveru, tak přímo ze zařízení. V případě zápisu jsou data vždy zapsána přímo do zařízení a operace by neměla skončit, dokud server

neověří, že data byla zařízením buď přijata, nebo odmítnuta. Z uvedeného vyplývá, že volání metod tohoto rozhraní může být poměrně časově náročné a není tudíž vhodné jej používat pro přenos velkých objemů dat nebo pro časté přístupy k serveru.

- **IOPCAsyncIO2** – (povinné rozhraní) slouží pro asynchronní čtení a zápis dat. Asynchronní v tomto případě znamená, že klient voláním metod tohoto rozhraní vznese požadavek na provedení operace, server tento požadavek zaznamená a metoda skončí. Server pak zajistí ve vhodném čase provedení vzneseného požadavku a informuje klienta o jeho úspěšnosti pomocí rozhraní IOPCDataCallBack, které klient zaregistroval na příslušném přípojném bodě. Klient má možnost přihlásit se k pravidelné aktualizaci hodnot všech položek ve skupině rovněž pomocí IOPCDataCallBack. Všechny čtení a zápisy hodnot jsou přímo ze zařízení, nikoliv z cache serveru.
- **IConnectionPointContainer** – (povinné rozhraní) poskytuje přístup k přípojnému bodu pro IOPCDataCallBack. Podrobnější popis je uveden v 2.2.3.
- **IEnumOPCItemAttributes** – (povinné rozhraní) poskytuje seznam všech položek definovaných ve skupině a hodnoty některých atributů této položky důležitých pro dané připojení.
- **IOPCAsyncIO** – jedná se o rozhraní sloužící pouze pro zachování zpětné kompatibility s OPC Data Access 1.0. Server, který nepodporuje tento standard, nemusí toto rozhraní implementovat.
- **IDataObject** – jedná se o rozhraní sloužící pouze pro zachování zpětné kompatibility s OPC Data Acess 1.0. Server, který nepodporuje tento standard, nemusí toto rozhraní implementovat.

3.2.5 IOPCDataCallBack

Každý klient, který chce používat asynchronní komunikaci se serverem (IOPCAsyncIO2), musí implementovat toto rozhraní. Zaregistruje je potom na příslušném přípojném bodě serveru, tak jak je popsáno v 2.2.3. Při vzniku některé z událostí (např. dokončení operace asynchronního zápisu dat) pak server zavolá příslušnou metodu tohoto rozhraní a tím informuje klienta o jejím vzniku. Jsou definovány čtyři metody, které pokrývají reakce na všechny podporované události.

- **OnDataChange** – pomocí této metody obdrží klient informaci o tom, že došlo ke změně hodnot položek ve skupině o větší hodnotu, než jaká je povolena parametrem Dead band. Metoda nikdy není volána častěji než je dáno parametrem Update rate. Další možnou příčinou volání metody je reakce na požadavek klienta o aktualizaci

hodnot všech položek v dané skupině. Klient obdrží informace o tom, jaká skupina událost vyvolala, jakých položek ve skupině se týká, a o nových hodnotách položek nebo případných chybách.

- **OnReadComplete** – oznamuje klientovi, že byl dokončen jeho požadavek o asynchronní čtení hodnot položek. Klient obdrží informace o tom, jaká skupina událost vyvolala, jakých položek ve skupině se týká, a o nových hodnotách položek nebo případných chybách. Pokud klient požadoval najednou čtení více položek, je tato metoda volána pouze jednou, po dokončení čtení všech položek.
- **OnWriteComplete** – oznamuje klientovi, že byl dokončen jeho požadavek o asynchronní zápis hodnot položek. Klient obdrží informace o skupině, která událost vyvolala, jaké položky byly zapsány a o chybách, které se při zápisu vyskytly. Pokud klient požadoval najednou zápis více položek, je tato metoda volána pouze jednou, po dokončení zápisu všech položek.
- **OnCancelComplete** – informuje klienta o úspěšném zrušení čekajícího požadavku na asynchronní operaci. Aby mohly být požadavky rušeny, jsou jim klientem přiřazovány identifikátory, podle kterých je možné operace odlišit.

3.3 Historical Data Access 1.0

3.3.1 Účel specifikace

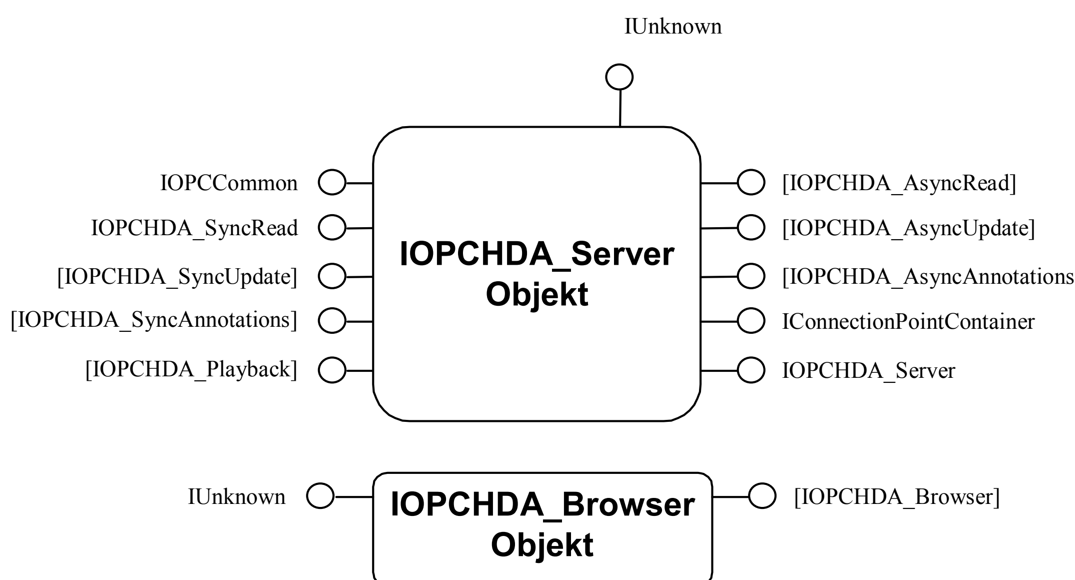
V současné době neustále narůstá množství dat dostupných na každé úrovni řídicích systémů. S využitím počítačových sítí a systémů pro přístup k datům v reálném čase (např. podle standardu popsaného v předchozí kapitole) lze jednoduše zajistit sběr těchto dat a jejich ukládání do databází. Je ovšem také nutné umožnit uživatelům zpracování a analýzu těchto „historických“ dat. Ta je důležitá jak pro management, protože může poskytnout důležité informace o volbě budoucích strategií, tak i pro inženýry řídicích systémů, kterým umožňuje jejich zkoumání a optimalizaci.

K dispozici je rostoucí počet databázových nástrojů umožňujících ukládání a analýzu historických dat. Až do doby vzniku této specifikace však neexistovala žádná možnost jak spojovat systémy od různých dodavatelů. OPC Historical Data Access (OPC HDA) specifikuje sadu komunikačních protokolů založených na standardu COM. Systémy podporující tuto specifikaci je možné spojovat ve smyslu Plug and Play. Je tedy možná interakce mezi jakýmkoliv serverem zpřístupňujícím historická data a klientem, který tato data vyžaduje.

Jsou definována rozhraní pro čtení, zápis i editaci historických dat. Protože velká část rozhraní je definovaná jako volitelná, umožňuje tento standard vytvářet datové servery různých úrovní. Jsou podporovány jak jednoduché servery, poskytující pouze přístup k surovým datům, tak i komplexní nástroje, umožňující ukládání, kompresi a analýzu dat.

3.3.2 Přehled objektů

Z hlediska této specifikace je model OPC HDA serveru logicky členěn do dvou objektů, jak je zobrazeno na obr. 3.7. Nepovinná rozhraní jsou zobrazena v hranatých závorkách. Je třeba upozornit, že se jedná pouze o logické členění, které nemusí mít žádnou spojitost se skutečnou implementací serveru.



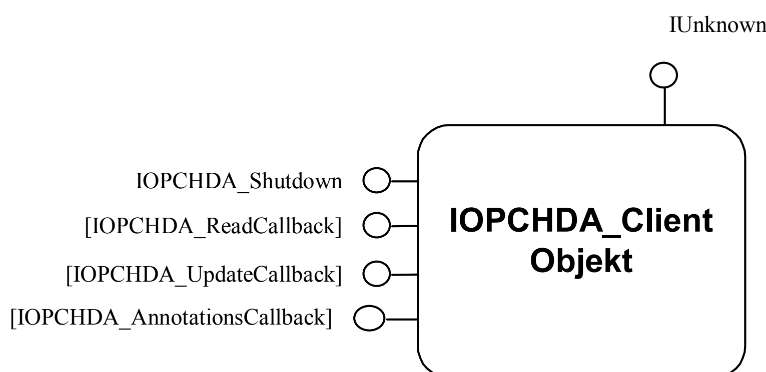
obr. 3.7 Model OPC HDA Serveru

Objekt Browser slouží pouze k prohlížení adresového prostoru serveru a poskytuje uživateli možnost rozhodnout se, které položky má zájem sledovat. Rozhraní IOPCHDA_Browser poskytuje podobné funkce jako rozhraní IOPCBrowseServerAddressSpace, popsané v 3.2.3. Je ale umístěno do zvláštního objektu, a proto má uživatel možnost používat najednou více těchto rozhraní a procházet tak nezávisle na sobě různé části adresového prostoru. To v případě OPC DA nebylo možné. Objekt server pak zajišťuje všechny další funkce spojené s přenosem historických dat.

V případě historických dat se nepředpokládá, že by uživatel četl najednou velké množství proměnných. Z toho důvodu zde neexistuje sdružování položek do skupin tak, jak tomu bylo v OPC DA. Klient může přistupovat přímo k jednotlivým položkám označovaným pomocí 32 bitových identifikátorů. Server poskytuje klientovi tyto identifikátory při procházení adresového

prostoru spolu se jmény položek. Klient určí pro každou položku svůj vlastní jedinečný identifikátor. Klient ve svých požadavcích musí vždy používat identifikátor dodaný serverem. Server naopak v návratových parametrech označuje jednotlivé položky identifikátory klienta. Tím je provedeno mapování mezi adresovým prostorem serveru a klienta.

Protože OPC HDA Server může podporovat několik rozhraní pro asynchronní komunikaci s klientem, je nutné, aby každý klient, který chce tato rozhraní využívat, implementoval odpovídající callback rozhraní. Model klienta pro OPC HDA je zobrazen na obr. 3.8. Nepovinná rozhraní jsou opět označena hranatými závorkami.



obr. 3.8 Model OPC HDA Klienta

3.3.3 Rozhraní objektu Server

Objekt server musí implementovat čtyři povinná rozhraní (kromě IUnknown) a může implementovat šest volitelných rozhraní pro rozšíření své funkčnosti. Uvedeme pouze stručný výčet rozhraní s popisem jejich účelu a funkce. Podrobnější informace lze nalézt v [7].

- **IOPCCommon –** (povinné rozhraní) viz 3.1.5.
- **IOPCHDAServer** – (povinné rozhraní) hlavní rozhraní serveru. Slouží pro zjištění obecných informací o serveru. Pomocí jeho metod lze získat informace o podporovaných agregačních funkcích a seznam definovaných atributů pro sledované položky spolu s popisy jejich významu.
- **IOPCHDA_SyncRead** – (povinné rozhraní) umožňuje synchronní čtení historických dat ze serveru. Data mohou být čtena pomocí několika různých metod popsaných v 3.3.4.
- **IOPCHDA_SyncUpdate** – (volitelné rozhraní) dává klientům možnost vkládat nová nebo přepisovat již existující data v databázi. Přepsaná data jsou nadále uchovávána v databázi. Jsou ale označena zvláštní značkou, takže je možné v případě potřeby obnovit původní stav databáze.

- **IOPCHDA_SyncAnnotations** – (volitelné rozhraní) umožňuje přístup k textovým popisům dat obsažených v jednotlivých položkách. Popisy jsou vždy označeny časovou značkou, takže je možné ukládat libovolné množství popisů pro každou položku a označit tak zvlášť různé zajímavé události v systému. Metody tohoto rozhraní umožňují jak vkládání, tak i čtení popisů.
- **IOPCHDA_AsyncRead** – (volitelné rozhraní) poskytuje uživateli metody pro asynchronní čtení dat. Jsou definovány obdobné funkce jako pro IOPCHDA_SyncRead. Kromě nich však existuje ještě další sada funkcí, pomocí nichž klient může žádat o čtení dat, která v současné době ještě nejsou k dispozici. Data jsou pak automaticky zasílána klientovi jakmile jsou dostupná.
- **IOPCHDA_AsyncUpdate** – (volitelné rozhraní) je asynchronní obdoba IOPCHDA_SyncUpdate.
- **IOPCHDA_AsyncAnnotations** – (volitelné rozhraní) je asynchronní obdoba IOPCHDA_SyncAnnotations.
- **IOPCHDA_Playback** – (volitelné rozhraní) umožňuje uživatelům asynchronní čtení uložených dat po částech. Uživatel nejprve získá data pro začátek požadovaného intervalu a zbývající data jsou mu postupně automaticky zasílána. Tím lze dosáhnout automatického přehrávání vývoje uložených dat.
- **IConnectionPointContainer** – (povinné rozhraní) poskytuje přístup k přípojným bodům pro všechna callback rozhraní. Podrobnější popis je uveden v 2.2.3.

3.3.4 Metody čtení dat

Všechna rozhraní serveru určená pro čtení dat podporují volitelně několik metod pro tento účel. Každá z metod provádí čtení odlišným způsobem, a proto zde popíšeme jejich funkce a rozdíly mezi nimi.

- **ReadRaw** – je základní metodou pro čtení dat, kterou musí podporovat každé rozhraní určené k tomuto účelu. Klient předá serveru začátek a konec časového intervalu o kterém si přeje informace, maximální počet vrácených hodnot a identifikátory položek, které chce číst. Server pak vrátí pro každou položku tři pole, ve kterých jsou uloženy hodnoty, časové známky a kvality. Jedná se přímo o záznamy uložené v databázi bez jakéhokoliv dalšího zpracování.

- **ReadProcessed** – opět umožňuje čtení hodnot položek ve specifikovaném časovém intervalu. Hodnoty jsou však před předáním klientovi zpracovány jednou z dostupných agregačních funkcí. Bližší popis tohoto zpracování je uveden v 3.3.5.
- **ReadAtTime** – při použití této metody předá klient serveru pole časových značek, pro které požaduje hodnoty specifikovaných položek. Pokud pro tyto časy hodnoty v databázi neexistují, zajistí server jejich lineární interpolaci ze dvou okolních hodnot.
- **ReadModified** – čte pouze data, která byla v databázi přepsána novými.

3.3.5 Agregační funkce

Pro analýzu historických dat je důležité, aby měl uživatel k dispozici kromě surových dat z databáze také jejich různé agregáty, jako průměry, extrémní hodnoty, počty hodnot za určený čas atd. Je samozřejmě mnohem efektivnější provádět tyto výpočty přímo na serveru a zmenšit tím objem přenášených dat ke klientovi.

Při použití agregačních funkcí musí klient specifikovat počátek a konec intervalu pro který mají být data načtena a vzorkovací interval. Pro každý vzorkovací interval v rámci daného časového období jsou pak všechna dostupná data přepočtena pomocí agregační funkce a výsledky jsou vráceny klientovi.

V rámci specifikace OPC HDA je definována široká škála agregačních funkcí, které mohou servery podporovat. Seznam a popis lze nalézt v [7]. Je však současně poskytnut i prostor pro definici vlastních funkcí specifických pro každý server.

4 Databáze procesních dat

Nedílnou součástí každého serveru historických dat je databáze, ve které jsou data ukládána. V této kapitole se budeme zabývat několika otázkami souvisejícími s návrhem databáze, která je předmětem této práce, a její realizací. Uvedeme některá pravidla, která je výhodné respektovat při návrhu struktury z důvodu optimalizace velikosti databáze a rychlosti přístupu k datům. Z hlediska použitelnosti a rozšiřitelnosti serveru je důležitá rovněž volba vhodného databázového stroje (serveru) a způsob připojení k tomuto serveru.

4.1.1 Použitý DB Server a připojení k serveru

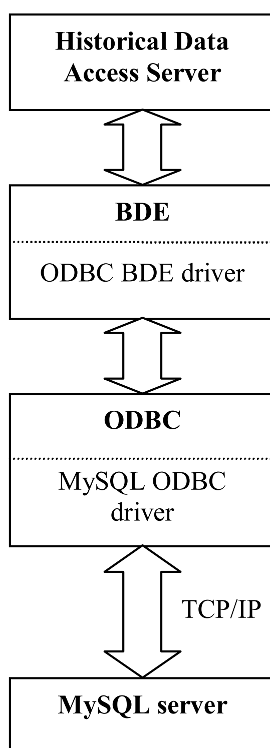
Databázový server je téměř vždy tvořen samostatnou aplikací, která není součástí serveru pro historická data. Z hlediska univerzálnosti použití a šiřitelnosti historického serveru je proto výhodné umožnit uživateli spojení s různými databázovými servery. Z tohoto důvodu byl zvolen přístup k databázovému serveru pomocí rozhraní ODBC (Open Database Connectivity). Jedná se o univerzální rozhraní implementované přímo do systému Windows, umožňující jakémukoliv klientovi přístup ke každému serveru podporujícímu toto rozhraní. Naprostá většina databázových serverů pak poskytuje ovladače pro připojení k ODBC.

Historical Data Access server je vytvořen v prostředí Borland C++ Builder s použitím sady integrovaných univerzálních databázových komponent usnadňujících práci s datovými zdroji. Firma Borland dodává se svými produkty rovněž aplikaci pro komunikaci s databázovými servery, nazvanou BDE (Borland Database Engine). Tento nástroj umožňuje propojení databázových komponent integrovaných v prosředí C++ Builder s databázovými servery. Pro každý server je ovšem potřeba použít specielní ovladače, které nemusí být vždy k dispozici. Proto byl přímo do BDE integrován ovladač pro propojení s rozhraním ODBC, který tak rozšiřuje použitelnost tohoto nástroje pro široké spektrum databázových serverů. Výsledná architektura propojení aplikace Historical Data Access server s databází je znázorněna na obr. 4.1.

Je patrné, že uvedený způsob propojení je poměrně složitý a každý použitý mezičlánek samozřejmě přináší jisté zpomalení přenosu dat mezi aplikací a databází. Na druhé straně přináší použitá metoda přístupu k databázi možnost použít vytvořenou aplikaci ve spolupráci s celou řadou databázových serverů.

Pro svoji práci jsem se rozhodl použít databázový server MySQL verze 3.23.54a Max. Jedná se o asi nejrozšířenější relační databázový server šířený jako „Open Source“ pod GNU veřejnou licencí (GPL). Je dostupný zdarma pro všechny zájemce jak v podobě zdrojových kódů,

tak jako přeložená funkční aplikace. Jeho tvůrcem je švédská firma MySQL AB. Server MySQL umožňuje svým uživatelům komunikaci pomocí standardizovaného jazyka SQL. I když nejsou implementovány všechny funkce vyžadované tímto standardem, jedná se o jeden z nejrychlejších dostupných SQL serverů.



obr. 4.1 Struktura propojení aplikace s databázovým serverem

Přestože k serveru MySQL jsou rovněž dostupné komunikační knihovny, které umožňují přímé spojení s aplikací vyvíjenou v C++, rozhodl jsem se pro použití ovladačů spojujících MySQL s ODBC a tím zachování možnosti propojení vytvořené aplikace i s jinými databázovými servery. Komunikace mezi ovladačem a serverem používá protokol TCP/IP a proto je možné použít vzdálené databázové servery. Tyto ovladače a jejich zdrojové kódy jsou rovněž dostupné v souladu s GPL.

Další informace o serveru a ovladačích lze nalézt v [13].

4.1.2 Požadavky na databázi

Nejdůležitější funkcí databáze v Historical Data Access serveru je samozřejmě uchovávání historických dat získávaných z různých zdrojů. Kromě samotných historických dat musí ale tento server poskytovat uživateli ještě řadu dalších informací, pro jejichž uložení je vhodné použít databázi. Konkrétní strukturou databáze a vazbami mezi jednotlivými tabulkami

se zabýváme v 4.1.3. V tomto odstavci se budeme věnovat obecným požadavkům, které musí databáze splňovat pro zajištění efektivního přístupu k datům a pro optimalizaci velikosti databáze.

Protože v databázi historických dat lze předpokládat sledování vývoje velkého množství položek z řídicích systémů za poměrně dlouhé časové úseky, je nutné počítat s velkým objemem uchovávaných dat. Proto musí být co nejvíce omezena redundance dat v databázi. Data musí být do databáze zaznamenávána v krátkých časových úsecích (v některých případech s intervalem jen několik milisekund) současně se však při čtení zpracovávají data za dlouhé časové úseky (velký objem dat). Je tedy nutné také optimalizovat rychlost přístupu k databázi. Těmto podmínkám nejlépe vyhovují relační databáze navržené ve třetí normální formě. Detailnější popis relačních databází a normálních forem databází je uveden například v [8]. Zde se omezíme pouze na stručný popis a vysvětlení uvedených pojmů.

Pojem relační databáze znamená, že data jsou ukládána v oddělených tabulkách spojených přesně definovanými relacemi, které umožňují spojování dat z jednotlivých tabulek. Takové uspořádání zvyšuje flexibilitu a rychlost databáze na rozdíl od přístupu, kdy jsou všechna data ukládána do jedné velké tabulky.

Normální formy jsou pak předpisy, které dále upřesňují, jak by měla vypadat struktura relační databáze, aby byla odstraněna redundance dat a zajištěna další optimalizace databáze.

- **První normální forma (1NF)** požaduje, aby jednotlivé atributy tabulek (sloupce), byly atomické, tedy dále nedělitelné. Pro každý atribut je tedy nutné zavést v příslušné tabulce vlastní sloupec. Dalším požadavkem je jedinečnost každého řádku tabulky (každé datové věty). Dva totožné řádky v jedné tabulce jsou zbytečnou redundancí. Posledním požadavkem pro databázi v 1NF je možnost přístupu k jednotlivým řádkům na základě obsahu klíčových atributů. V každé tabulce je tedy několik atributů, které jsou označeny jako klíčové a každý řádek tabulky musí být jednoznačně identifikován pomocí unikátní kombinace hodnot těchto atributů.
- **Druhá normální forma (2NF)** požaduje, aby daná relační databáze byla v první normální formě a zároveň žádný atribut nesmí být parciálně funkčně závislý na klíči příslušné tabulky. Jinými slovy: pokud jsou hodnoty jednoho ze sloupců tabulky, který není obsažen v klíči, závislé pouze na některých sloupcích klíče, je nutné takovou tabulku rozdělit na několik dílčích. Jako příklad uveďme následující tabulku:

Zařízení	Jméno položky	Datový typ
Motor	Otáčky	Integer
Motor	Stav oleje	Bool
Katalyzátor	Teplota	Double

Klíčem v této tabulce jsou atributy Zařízení a Jméno položky. Tabulka není ve 2NF protože atribut datový typ je závislý pouze na atributu jméno položky a tedy pouze na části klíče. Pro převod do 2NF je nutné tabulku rozdělit a upravit následujícím způsobem:

Zařízení	Jméno položky
Motor	Otáčky
Motor	Stav oleje
Katalyzátor	Teplota

Jméno položky	Datový typ
Otáčky	integer
Stav oleje	bool
Teplota	double

- **Třetí normální forma (3NF)** požaduje, aby daná relační databáze byla ve 2NF a zároveň aby žádný neklíčový atribut nebyl transitivně závislý na klíči. Tedy žádný neklíčový atribut nesmí být závislý na jiném neklíčovém atributu, který je závislý na klíči. Jako příklad opět uveďme tabulku (tentokrát pouze její definici):

Jméno položky	Jméno zařízení	Umístění zařízení

Tato tabulka není ve 3NF, protože atribut Umístění zařízení je závislý na Jménu zařízení, které je závislé na Jménu položky. Normalizaci v tomto případě opět provedeme pomocí rozdělení do dvou tabulek s následujícími atributy:

Jméno položky	Jméno zařízení

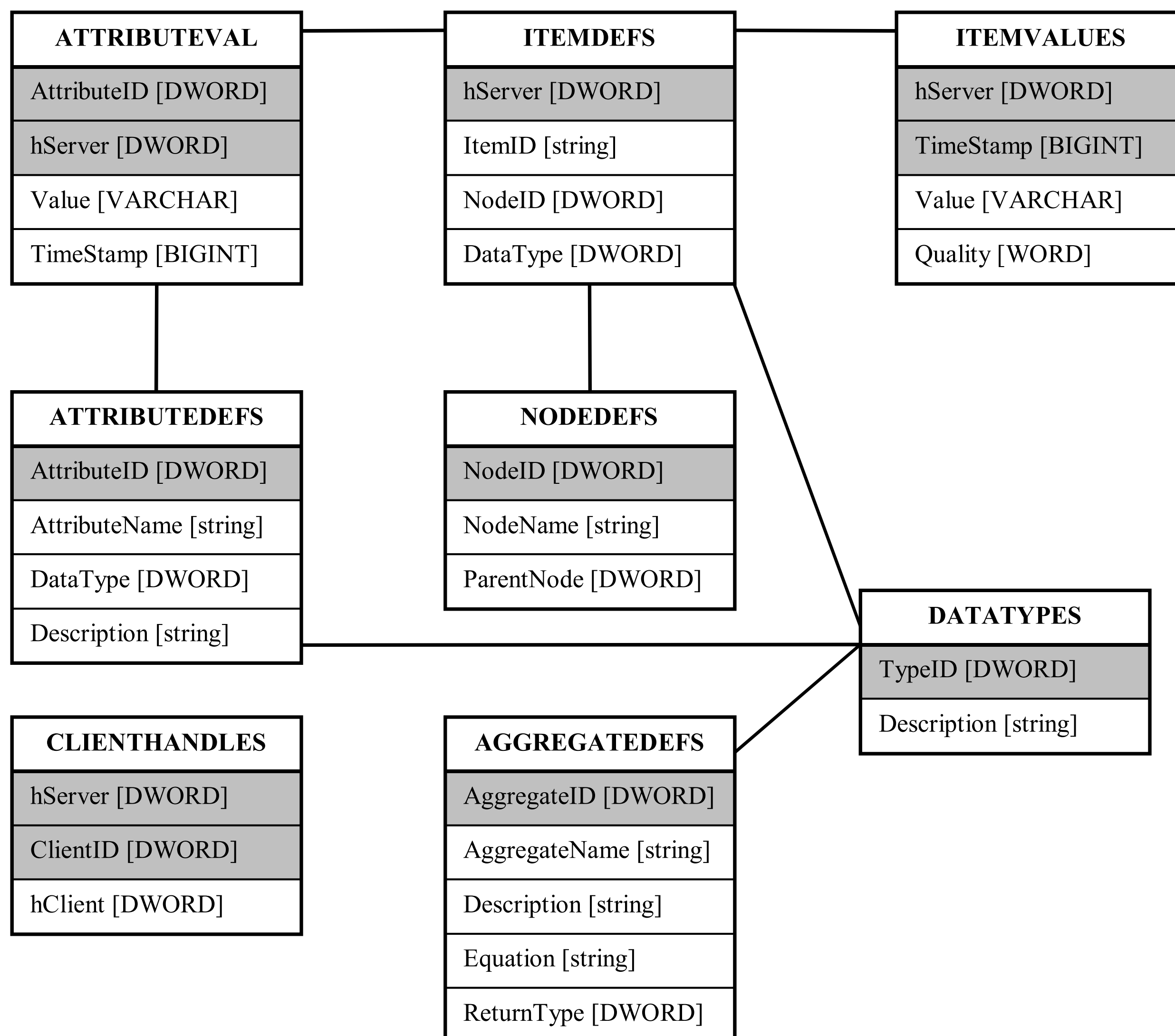
Jméno zařízení	Umístění zařízení

Databáze ve třetí normální formě je optimalizovaná pro možnost rychlého, přehledného a flexibilního ukládání velkých množství dat. V případě komplikovaných dotazů je na druhé straně nutné propojení mnoha tabulek, což může prodloužit dobu provádění dotazů.

4.1.3 Struktura databáze

V Historical Data Access serveru, implementovaném v rámci této diplomové práce, je databáze použita nejen pro ukládání hodnot jednotlivých sledovaných položek, ale také pro mnoho dalších informací, které musí OPC HDA server poskytovat. Celková struktura databáze s výčtem všech tabulek a jejich atributů je znázorněna na obr. 4.2. Pro každou tabulku jsou

atributy tvořící klíč zvýrazněny.Příkazy v jazyce SQL, umožňující vytvoření těchto databází, jsou uvedeny v Příloze A. Rozeberme nyní podrobněji jednotlivé tabulky, jejich atributy a vztahy s ostatními tabulkami v databázi.



obr. 4.2 Struktura databáze Historical Data Access serveru

- **ITEMDEFS** – v této tabulce jsou obsažena data definující jednotlivé položky sledované serverem. Každá položka je jednoznačně určena pomocí jejího identifikátoru, nazvaného v tomto případě hServer. Pro každou položku je zde rovněž uveden její jednoznačný textový identifikátor (ItemID), datový typ (DataType) a identifikátor uzlu datového prostoru serveru (NodeID), ke kterému položka přísluší. Příklad definice adresového prostoru serveru je uveden v příloze F.
- **ITEMVALUES** – zde jsou uložena historická data definovaných položek. Data jsou zaznamenávána ve formě trojice: Hodnota (Value) , Čas (TimeStamp) a Kvalita (Quality). U

každého záznamu je samozřejmě rovněž identifikátor položky, které se záznam týká (hServer). Protože identifikátor položky a čas tvoří klíč této tabulky, nemůže zde být obsaženo více záznamů hodnot jedné položky v jednom čase. Na druhou stranu je takovouto volbou indexu velmi urychlen přístup do tabulky, která může obsahovat velký počet záznamů.

Vzhledem k tomu, že v řešené tabulce je potřeba ukládat hodnoty různých typů, zvolil jsem pro uložení hodnoty atribut typu VARCHAR s maximální délkou 250 znaků. Tento atribut umožňuje uložení hodnot různé délky, přičemž v cílové struktuře je vždy rezervováno místo podle skutečné délky příslušného záznamu. Číselné typy jsou tedy při ukládání převáděny na řetězec, ale při použití vestavěných agregačních funkcí serveru je s nimi nakládáno jako s číselnými hodnotami. Při převodu na řetězec nedochází u čísel s plovoucí desetinou čárkou ke ztrátě přesnosti, díky velkému počtu uložených znaků. Nároky na velikost cílové struktury jsou však samozřejmě větší, než by byly v případě uložení v binární podobě.

Ukládání časových značek je věnován zvláštní odstavec 4.1.4.

- **NODEDEFS –** tato tabulka vytváří druhou část definice adresového prostoru serveru. Jsou zde uloženy informace o uzlech rozdělujících adresový prostor hierarchicky na logické části. Každý uzel je jednoznačně určen pomocí identifikátoru (NodeID). Je mu rovněž přiřazeno textové označení, pod kterým je prezentován uživateli (NodeName). Pro vytvoření hierarchického prostoru musí mít každý uzel rovněž určen svůj nadřazený uzel (ParentNode). Tato tabulka musí vždy obsahovat alespoň jeden záznam definující takzvaný kořen stromové struktury. Identifikátor tohoto uzlu je roven jedné a nadřazený uzel je označen jako nula. Tím je určeno, že se jedná o kořenový uzel. Každá databáze může mít pouze jeden kořenový uzel. Příklad definice adresového prostoru serveru je uveden v příloze F.
- **ATTRIBUTEDEFS –** zde jsou definovány atributy (vlastnosti), které je možné v databázi uchovávat pro každou položku. Jak již bylo zmíněno, slouží atributy k poskytnutí dodatečných informací o položce. Podle standardu OPC HDA musí být každý atribut označen jednoznačným identifikátorem (AttributeID), jménem (AttributeName) a musí pro něj být dostupný popis jeho významu (Description). Dále je v této tabulce rovněž definován datový typ atributu (DataType).

  Pokud uživatel přidá pomocí vytvořeného konfiguračního nástroje (viz 5.2.4) definici nové položky, je do tabulky ATTRIBUTEVAL automaticky přidán příslušný počet záznamů a vyplněny příslušné identifikátory. Seznam implicitně definovaných atributů pro vytvořený server a jejich význam je uveden v Příloze B.

- **ATTRIBUTEVAL** – uchovává hodnoty definovaných atributů pro definované položky. Každý záznam je jednoznačně identifikován pomocí identifikátoru příslušného atributu (AttributeID) a identifikátoru položky, ke které se záznam vztahuje (hServer). Ukládání hodnot (Value) a časových značek (TimeStamp) je podobné jako v tabulce ITEMVALUES, popsané výše.

  Současná verze serveru podporuje pouze jedinou hodnotu každého atributu pro jednotlivé položky.
- **AGGREGATEDEFS** – záznamy v této tabulce definují agregační funkce podporované serverem. Podrobnější popis způsobu definice agregačních funkcí a jejich výpočtu je uveden v 4.1.5. Každá podporovaná agregační funkce je v souladu se specifikací označena identifikátorem (AggregateID), jménem (AggregateName) a textovým popisem své funkce (Description). Předpis pro výpočet je uložen v atributu Equation. Protože agregační funkce mohou vracet i hodnoty jiných typů, než jakých jsou původní data (například funkce Count vždy vrací číslo typu integer), je tento typ specifikován v atributu ReturnType. V případě, že funkce vrací hodnoty stejného typu, jako jsou původní data, je hodnota tohoto atributu nastavena na nulu. Seznam implicitně definovaných agregačních funkcí, jejich popis a předpisy pro jejich výpočet jsou uvedeny v Příloze C.
- **DATATYPES** – v této tabulce jsou uvedeny popisy (Description) a identifikátory (TypeID) všech datových typů podporovaných specifikací OPC HDA.
- **CLIENTHANDLES** – tabulka umožňující mapování mezi adresovými prostory klientů a serveru tak, jak je uvedeno v 3.3.2. Každý klient má možnost specifikovat svůj vlastní identifikátor (hClient) pro všechny položky sledované serverem, určené pomocí identifikátoru serveru (hServer). Klient je specifikován pomocí atributu ClientID, který je ukazatelem na COM objekt třídy IOPCHDA_Server obsluhující dané spojení s klientem. Při komunikaci mezi klientem a serverem označuje klient položky pomocí identifikátoru serveru a server předává klientovi položky označené identifikátorem klienta.

4.1.4 Ukládání časových značek

Časové značky jsou velmi důležitou a nedílnou součástí uložených dat. Ve všech OPC specifikacích se časové značky pro jednotlivé hodnoty předávají pomocí Windows API struktury FILETIME. Tato struktura je definována jako 64 bitový integer, který udává čas relativně. Číslo představuje počet stovek nanosekund uplynulých od počátku kalendáře. Se strukturou je možné pracovat také jako se dvěma 32 bitovými čísly. Tento způsob minimalizuje objem uložených dat a přitom zachovává dostatečnou přesnost pro všechny aplikace.

Databázové servery nabízejí několik formátů ukládání času do databáze. Žádný z těchto formátů však neposkytuje dostatečnou přesnost (jednotky milisekund), a proto jsem se rozhodl ukládat časové značky do databáze přímo v původním formátu. Takto uložené časy nejsou pro uživatele, prohlížejícího obsah databáze jinak než pomocí serveru, příliš srozumitelné, nabízejí ale několik podstatných výhod.

Data postupně zapisovaná do databáze tvoří svými časovými známkami vzestupnou řadu a tím je velmi usnadněna indexace záznamů v tabulkách. Indexování podle času je pro hodnoty důležité, protože velmi urychluje přístup k datům pro velké množství záznamů. Další výhodou je jednoduché vytváření skupin hodnot tvořících jednotlivé vzorkovací intervaly při výpočtu agregačních funkcí. Výpočet agregačních funkcí je blíže popsán v 4.1.5.

Jedním z problémů, vyskytujících se při záznamu dat přes delší časové období, je přechod mezi zimním a letním časem, způsobující nekonzistentnost časových známek v databázi. V dostupných OPC specifikacích neexistují žádná pravidla pro řešení tohoto problému a jeho implementace je tedy na tvůrci systému.

Nabízí se několik možností. První způsob je ukládat do databáze skutečné časové známky respektující změnu času. V tom případě je třeba zvláštním způsobem ošetřit ukládání dat při přechodu z letního na zimní čas, kdy by došlo ke ztrátě dat v úseku jedné hodiny. V souladu se specifikací OPC HDA je možné řešit to uložením více údajů pro jeden časový okamžik. Uživatel je pak na tuto skutečnost upozorněn při čtení dat návratovým kódem EXTRA_DATA a má možnost pomocí metody *ReadModified* získat přepsaná data. Při použití tohoto přístupu není nutné, aby klient vyžadující data ze systému, prováděl úpravu časových značek. Problém ovšem nastává při výpočtu agregací dat přes časový úsek zahrnující okamžik přechodu z jednoho časového formátu na druhý. Správné načtení dat pak musí být zvlášť ošetřeno agregační funkcí.

Druhým přístupem je ukládání dat do databáze s časovými známkami nereflektujícími přechod na letní čas. Při použití tohoto postupu odpadají všechny problémy s výpočtem agregačních funkcí a zajištěním proti ztrátě dat v okamžiku přechodu. Na druhé straně je pak nutné při čtení dat ať už na straně serveru nebo klienta provést příslušnou úpravu časových značek tak, aby odpovídaly skutečnosti.

V implementovaném serveru jsou pro všechny hodnoty ukládány skutečné časové značky, jak jsou poskytovány systémy zpřístupňujícími technologická data. Databáze ale neumožňuje uložení více hodnot jedné položky pro jednu časovou značku (díky volbě klíče v tabulce hodnot), a proto není ošetřeno zabezpečení proti ztrátě dat. Při změně času o jednu hodinu zpět (přechod ze zimního na letní čas) může dojít k přepsání hodnot se stejnými časovými známkami a tím ke ztrátě jedné hodiny dat.

4.1.5 Definice a metoda výpočtu agregačních funkcí

Agregační funkce jsou velmi mocným a důležitým nástrojem serveru pro historická data. Dávají uživateli bohaté možnosti analýzy uložených dat. Z tohoto důvodu je vhodné, aby sada těchto funkcí byla modifikovatelná a rozšiřitelná podle požadavků uživatele. Rozhodl jsem se tedy zahrnout definice agregačních funkcí přímo do databáze serveru a tím umožnit snadnou úpravu již definovaných funkcí nebo přidání nových. Agregační funkce jsou definovány v tabulce AGGREGATEDEFS popsané v 4.1.3.

Protože každý databázový server poskytuje širokou škálu vestavěných agregačních funkcí (standardizovaných jazykem SQL), jsou tyto použity pro výpočet agregací historických dat. Každá funkce je definována pomocí textové podoby SQL příkazu SELECT, obsahujícího parametry udávající identifikátor položky, které se výpočet týká, a příslušné časové úseky. Při volání metody *ReadProcessed* je pak podle identifikátoru požadované agregační funkce nalezen příslušný záznam v tabulce. Parametry v příkazu SELECT jsou nahrazeny skutečnými hodnotami parametrů funkce a příkaz je předán ke zpracování serveru. Výsledek je pak navrácen klientovi.

Příkaz definující implementované funkce má obecně následující strukturu:

```
SELECT AggrFunc(Value) as Value, MIN(TimeStamp) as TimeStamp, Quality
FROM ITEMVALUES
WHERE hServer = :hServer AND
      TimeStamp >= :Start AND
      TimeStamp <= :End
GROUP BY (ROUND((TimeStamp-:Start)/:Resample))
LIMIT :Limit
```

Parametry funkce jsou označeny pomocí znaku : a mají následující význam:

`:hServer` identifikuje položku, které se agregační funkce týká,

`:Start` určuje počátek intervalu, pro který se počítá agregační funkce,

`:End` určuje konec intervalu, pro který se počítá agregační funkce,

`:Resample` udává délku vzorkovacího intervalu,

`:Limit` omezuje maximální množství výstupních záznamů.

Klauzule GROUP BY zajišťuje rozdělení zkoumaného intervalu na vzorkovací intervaly. Podle definice příkazu SELECT se agregační funkce počítají zvlášť pro každou skupinu záznamů, které mají shodnou hodnotu výrazu uvedeného v části GROUP BY.

Je zřejmé, že pomocí této struktury lze definovat jen omezenou množinu agregačních funkcí (seznam všech implementovaných funkcí je možné nalézt v Příloze C). Hlavní omezení spočívá v limitované množině agregačních funkcí definovaných jazykem SQL. Pokud by

uživatel požadoval výpočet složitějších funkcí, bylo by nutné zvolit jinou metodu. Jedním z řešení je využití procedur vložených v databázi, které jsou volány databázovými klienty a umožňují výpočet složitějších výrazů (server MySQL bohužel zatím tuto funkci nepodporuje). Další možností, která se nabízí, je využití externích programů pro výpočet. V tom je možné s výhodou opět využít COM rozhraní. Výpočet těchto externích agregací by mohl probíhat jak na straně serveru, v rámci volání metody *ReadProcessed*, tak i na straně klienta, který by vstupní data získal například pomocí metody *ReadRaw*.

S použitím programu ActiveX HDA Client (viz 5.7) je možné přenášet data mezi HDA serverem a jakoukoliv aplikací umožňující práci s ActiveX objekty, například Matlab nebo MS Excel. V těchto aplikacích pak lze provádět se získanými daty i velmi složité výpočty a následnou vizualizaci.

5 Popis aplikací

V této kapitole se budeme zabývat jednotlivými aplikacemi vytvořenými v rámci této diplomové práce. Pro každou aplikaci vždy uvedeme její účel a popis funkce. Dále pak popis důležitých tříd implementovaných v této aplikaci. Podrobnosti týkající se jednotlivých programů je možné nalézt ve zdrojových kódech uložených na přiloženém CD-ROM.

Všechny programy byly vytvořeny v jazyce C++ s použitím prostředí Borland C++ Builder 5.0. V maximální možné míře jsou použity objekty dodávaných knihoven. Objekty z knihoven VCL (Visual Component Library) jsou použity především pro grafické rozhraní aplikací a pro připojení k databázovému serveru. Jejich bližší popis je možné nalézt především v dokumentaci k Borland C++ Builder. Pro práci s COM je použita knihovna ATL (Automation Library) vyvinutá původně firmou Microsoft a upravená pro C++ Builder.

5.1 OPC Client

První vytvořenou aplikací, sloužící především pro seznámení s technologií OPC a pro testovací účely, je jednoduchý klient pro rozhraní OPC Data Access 2.0. Tento klient umožňuje připojení k lokálnímu i vzdálenému serveru podporujícímu toto rozhraní. Je možné navázat spojení s libovolným počtem serverů. V rámci každého spojení lze vytvářet skupiny a pomocí grafického rozhraní přidávat do těchto skupin položky poskytované serverem. Hodnoty položek jsou pak zobrazeny v tabulkové podobě a automaticky aktualizovány se zadanou frekvencí.

Klient je určen především pro testování synchronního i asynchronního přenosu dat ze serveru. Skupina objektů určených pro komunikaci se serverem je pak využita v dalších aplikacích, především pak pro záznam dat do databáze pomocí programu DB OPC Client (viz. 5.5).

5.1.1 Objekty pro OPC DA

Jedním z cílů této aplikace bylo vytvořit a otestovat sadu objektů, umožňujících jednoduché a transparentní spojení s jakýmkoliv OPC DA serverem a využitelných i bez detailnějších znalostí OPC specifikací. Všechny objekty jsou určeny pro zapouzdření jednotlivých úrovní komunikace a musí zajistit ošetření všech chybových stavů. Uveďme nyní jejich podrobnější popis.

TOPCServerConnection – tento objekt zapouzdřuje na straně klienta všechny funkce poskytované COM objektem OPC Server. Především zajišťuje inicializaci COM rozhraní a poskytuje seznam ProgID všech OPC serverů instalovaných na počítači specifikovaném jménem

uzlu (Node). Metoda *ConnectServer* naváže spojení se serverem se zadaným ProgID. Další metody pak umožňují prohlížení adresového prostoru serveru. Definované větve (Branches) a položky (Leaves), stejně tak jako jména instalovaných serverů, jsou předávány jako seznam řetězců (objekt TStringList), což usnadňuje spolupráci s grafickými objekty (ComboBox, ListBox …). Je možné také vytvářet nové objekty *TOPCGroupConnection* (viz. níže) se zadanými parametry.

tabulka 5-A Seznam vybraných metod třídy TOPCServerConnection

Deklarace metody	**Popis**
`bool ConnectServer(AnsiString fServerID,AnsiString fNodeName)`	Naváže spojení se serverem specifikovaným jeho ProgID v parametru *fServerID* a jménem počítače (uzlu).
`bool DisconnectServer(void)`	Ukončí spojení se serverem.
`COPCGroupClient * CreateGroup ( AnsiString GroupName, bool Active, DWORD UpdateRate, float DeadBand)`	Vytvoří objekt typu TOPCGroupConnection se zadanými parametry (význam parametrů je popsán v odstavci 3.2.2.
`TStringList * GetBranchNames()`	Vrátí seznam větví (uzlů) definovaných na aktuální pozici v adresovém prostoru serveru.
`TStringList * GetLeafNames()`	Vrátí seznam listů (položek) definovaných na aktuální pozici v adresovém prostoru serveru.
`TStringList * GetLeafIDs(TStringList * LeafNames)`	Vrátí seznam identifikátorů (hServer) položek definovaných jejich jmény.
`bool BrowseUp (void)`	Změní aktuální pozici v adresovém prostoru na nadřazenou úroveň.
`bool BrowseDown (WideString NodeName)`	Změní aktuální pozici v adresovém prostoru na nižší úroveň specifikovanou jménem uzlu (větve).
`TStringList * GetServerNames( AnsiString NodeName)`	Vrátí seznam ProgID všech OPC DA serverů nainstalovaných na počítači specifikovaném pomocí *NodeName*.

TOPCGroupConnection je zapouzdřením COM objektu OPC Group a zpřístupňuje jeho funkce. Většinou není vytvářen přímo pomocí volání svého konstruktoru, ale jednou z metod objektu TOPCServerConnection. Objekt slouží především jako kontejner pro objekty

TOPCItemConnection a pro jejich vytváření a rušení. Zpřístupňuje seznam jmen všech obsažených položek a ukazatele na příslušné objekty. Dále pak obsahuje metody umožňující hromadné čtení a zápis hodnot všech položek obsažených ve skupině, jak synchronní, tak asynchronní. Pro příjem asynchronních událostí jsou definovány čtyři události (Event), korespondující ke každé z metod definovaných na rozhraní IOPCDataCallBack (3.2.5). Na tyto události mohou být napojeny uživatelem definované metody. Před voláním uživatelských metod jsou nejprve aktualizovány hodnoty v příslušných objektech *TOPCItemConnection*. Parametry nesou informace o počtu změněných položek a ukazatele na jejich objekty.

tabulka 5-B Seznam vybraných metod třídy TOPCGroupConnection

Deklarace metody	**Popis**
`COPCItemClient * AddItem(AnsiString ItemID, bool Active, bool DirectRead, bool DirectWrite, int Handle)`	Vytvoří nový objekt typu TOPCItemConnection se specifikovanými parametry a přidá jej do vnitřního seznamu.
`bool RemoveItem(AnsiString ItemID)`	Odebere z vnitřního seznamu objekt TOPCItemConnection specifikovaný jeho jménem.
`COPCItemClient * GetItem(AnsiString ItemID)`	Vrátí ukazatel na objekt TOPCItemConnection specifikovaný jeho jménem.
`void SyncReadAllItems()`	Provede čtení dat všech položek ve skupině pomocí rozhraní IOPCSyncIO.
`void SyncWriteAllItems()`	Provede zápis dat všech položek ve skupině pomocí rozhraní IOPCSyncIO.
`void AsyncReadAllItems()`	Provede čtení dat všech položek ve skupině pomocí rozhraní IOPCAsyncIO2.
`void AsyncWriteAllItems()`	Provede zápis dat všech položek ve skupině pomocí rozhraní IOPCAsyncIO2.

TOPCItemConnection představuje nejnižší stupeň v hierarchii objektů. Udržuje v sobě informace o příslušné položce, jako její jméno, handle a především pak aktuální hodnotu, včetně časové známky a kvality. Hodnota je uchovávána pomocí objektu Variant a tak jsou podporovány položky všech základních typů. Kvalita i časová známka jsou automaticky převáděny do textové podoby, čas je možné rovněž získat jako objekt TDateTime používaný

standardně v prostředí C++ Builder. Objekt obsahuje i metody pro čtení a zápis hodnot jednotlivých položek, ty však využívají služeb rozhraní IOPCSyncIO a IOPCAsyncIO2 zapouzdřených objektem TOPCGroupConnection.

tabulka 5-C Seznam vybraných metod a vlastností třídy TOPCItemConnection

Deklarace	**Popis**
`void SyncRead()`	Načte hodnotu položky pomocí IOPCSyncIO.
`void SyncWrite()`	Zapíše hodnotu položky pomocí IOPCSyncIO.
`void AsyncRead()`	Načte hodnotu položky pomocí IOPCAsyncIO.
`void AsyncWrite()`	Zapíše hodnotu položky pomocí IOPCAsyncIO.
`Variant Value`	Aktuální hodnota položky. Při zápisu nové hodnoty do této vlastnosti může být hodnota přímo zapsána do OPC serveru nebo pouze uložena v rámci třídy a zapsána později pomocí metod *SyncWrite* nebo *AsyncWrite.*
`AnsiString ID`	Jméno položky. Pod tímto jménem je položka registrována v objektu TOPCGroupConnection.
`FILETIME TimeStamp`	Aktuální časová známka pro hodnotu uloženou ve *Value*. Pouze pro čtení.
`WORD Quality`	Specifikace kvality pro hodnotu uloženou ve *Value*. Pouze pro čtení.

5.1.2 Grafická část aplikace

Grafické rozhraní bylo sestaveno s pomocí objektů knihoven VCL. Pro získávání hodnot z OPC využívá výše popsané objekty, které poskytují data ve formátech podporovaných VCL objekty. Aplikace je typu MDI (Multiple Document Interface), pro každé spojení se serverem je vytvořen nový formulář. V prostoru formuláře je pak zobrazen seznam vytvořených skupin a pro vybranou skupinu v tabulkové podobě jména definovaných položek, jejich hodnoty, časové známky a kvality. Při každém vyvolání jedné z událostí definovaných v objektu TOPCGroupConnection jsou pak zobrazené údaje automaticky obnoveny.

Je možné zvolit automatické obnovování hodnot položek se zvolenou frekvencí a pásmem necitlivosti tak, jak je definováno ve standardu OPC DA. Čtení a zápis hodnot lze provádět i manuálně pro celou skupinu, nebo pro každou položku zvlášť.

5.2 HDA Server

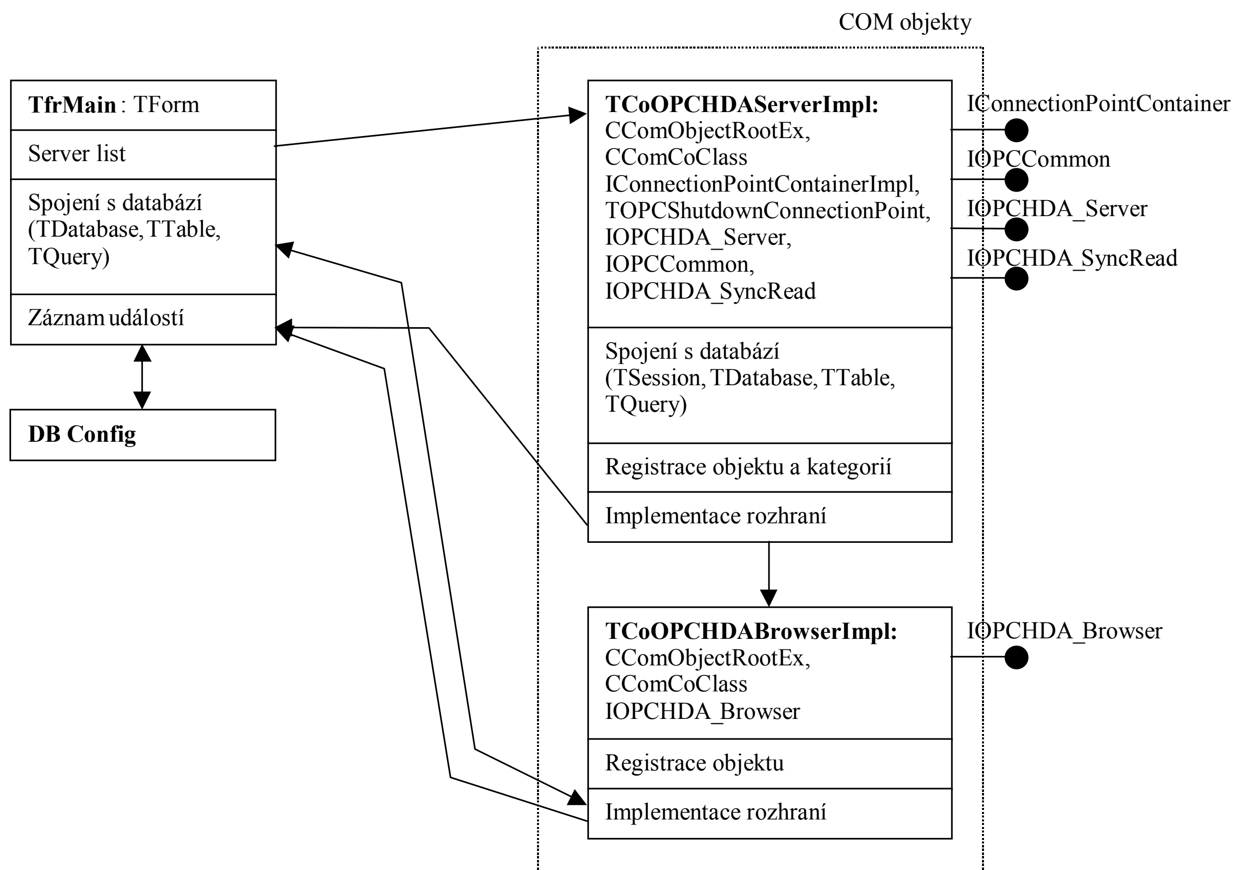
Historical Data Access Server vyhovující standardu OPC HDA 1.0 byl hlavní náplní této diplomové práce. Uvedená specifikace ponechává do značné míry na programátorovi volbu jaké funkce bude server podporovat. Rozhodl jsem se proto pro tvorbu serveru podporujícího pouze synchronní čtení dat z databáze. Server tedy implementuje všechna povinná rozhraní definovaná specifikací a dále rozhraní IOPCHDASyncRead (blíže popsané v 3.3.3). Všechna další nepovinná rozhraní nejsou implementována. Na rozhraní IOPCHDASyncRead jsou implementovány všechny definované metody.

5.2.1 Struktura aplikace

HDA Server se skládá ze dvou hlavních částí. První část představuje samotný OPC server tak, jak je definován specifikací, a jeho grafické rozhraní, umožňující administrátorovi serveru volbu databáze s procesními daty a nastavení záznamu prováděných operací do souboru (log). Druhá část je konfigurační databázový nástroj pro definici adresového prostoru serveru. Ten je blíže popsán v odstavci 5.2.4.

HDA server musí implementovat dva COM objekty obsahující příslušná rozhraní tak, jak je popsáno v 3.3.2. Při návrhu objektového modelu aplikace jsem využil tohoto logického členění. Vytvořil jsem dvě třídy definující všechny metody požadované specifikací pro uvedené objekty. Tyto třídy jsou potomky několika rodičovských tříd z knihoven ATL, které zajišťují základní funkce společné pro všechny COM objekty. Dále pak dědí své vlastnosti od všech implementovaných rozhraní tak, jak je požadováno COM modelem.

Celková struktura aplikace je zobrazena na obr. 5.1. Jsou zde zobrazeny nejdůležitější objekty aplikace, jejich hlavní části a vztah mezi nimi. Objekt TfrMain představuje především grafickou část aplikace. Dále pak obsahuje VCL objekty pro spojení s databází a rutiny pro záznam událostí do souboru. Jsou zaznamenávány všechny události přicházející od dalších objektů aplikace. Pro každé klientské spojení je proveden záznam kdy byla volána některá z metod rozhraní a výsledek tohoto volání. Každý záznam obsahuje časovou známku a identifikátor klienta, který danou událost vyvolal. Databázové spojení je využíváno všemi objekty TCoOPCHDABrowserImpl.



obr. 5.1 Objektová struktura aplikace HDA Server a hlavní části objektů

Objekty TCoOPCHDAServerImpl a TCoOPCHDABrowserImpl představují COM objekty, které jsou vytvářeny pro každé spojení s klientem. Je pro ně proto nutná volba modelu tvorby vláken (viz 2.1.11). TCoOPCHDABrowserImpl je určen pro procházení adresového prostoru serveru. Tato operace není nijak časově náročná a nepředpokládá se časté volání metod tohoto objektu klienty. Z těchto důvodů jsem pro zjednodušení aplikace zvolil model Single. Všechny objekty tohoto typu pro všechna spojení jsou proto spouštěny v hlavním (VCL) vlákně aplikace a mohou tedy bez problémů využívat spojení s databází vytvořené v rámci TfrMain.

TCoOPCHDAServerImpl je naproti tomu nejdůležitějším objektem spojení s klientem, který zajišťuje čtení dat z databáze a přenos ke klientovi. Některé metody tohoto objektu mohou být velmi časově náročné (např. *ReadProcessed*), a proto je vhodné, aby byl každý z těchto objektů spouštěn ve zvláštním vlákně a neblokoval tak provádění ostatních operací. Zvolil jsem tedy model Apartment. Tento model přináší především pro lokální servery značné zlepšení výkonu oproti modelu Single a zachovává přitom některé vlastnosti usnadňující tvorbu aplikace. Je už ale nutné, aby každý z těchto objektů disponoval vlastním spojením s databází, protože databázové operace zabírají naprostou většinu času nutného pro dokončení volání metod objektu. Pro záznam událostí o volání metod mohou být bezpečně využívány metody objektu TfrMain,

protože synchronizace s VCL vláknem probíhá automaticky a tyto operace nejsou časově náročné. Podrobněji se popisem těchto objektů budeme zabývat v odstavcích 5.2.2 a 5.2.3.

Technologie spojení s databází je blíže popsána ve 4.1.1. BDE objekty (TSession, TDatabase, TTable a TQuery) pak použití tohoto spojení velmi usnadňují. Jejich bližší popis lze nalézt v [2]. Jak bylo uvedeno výše, je pro každé spojení s klientem vytvořeno nové vlákno a s ním i nové spojení s DB serverem. Databázový server pak pro každé připojení vytváří rovněž nové vlákno, a proto jsou jednotlivé požadavky od klientů zpracovávány zcela nezávisle.

5.2.2 Třída CoOPCHDAServerImpl

Objekty této třídy zajišťují veškerou funkčnost vyžadovanou rozhraními definovanými ve specifikaci OPC HDA pro objekt IOPCHDA_Server. Protože mnoho vlastností a funkcí je společných pro většinu COM objektů, byly v rámci knihovny ATL vytvořeny třídy implementující tyto vlastnosti. Nově vytvářená třída pak může tyto vlastnosti zdědit. Jak je patrné z obr. 5.1, použil jsem při vytváření CoOPCHDAServerImpl několik předků z knihoven ATL, a proto zde stručně uvedu jejich funkci. Bližší popis je možné nalézt v [11].

CComObjectRootEx obsahuje metody COM objektu ClassCreator a zajišťuje tak všechny potřebné funkce pro automatické vytváření objektů pomocí COM rozhraní. Třída je deklarována jako šablona (template). Parametrem pro rozvinutí šablony je specifikace modelu vláken požadovaného pro vytvářený objekt.

CComCoClass obsahuje především definici všech funkcí rozhraní IUnknown. Opět se jedná o šablonu. Při jejím rozvíjení jsou pomocí speciálních maker získány informace o všech rozhraních definovaných ve vytvářeném objektu. Ty jsou nutné pro implementaci metody *QueryInterface*.

IConnectionPointContainerImpl implementuje rozhraní IConnectionPointContainer (viz 2.2.3). Opět jsou pomocí sady maker získány informace o všech deklarovaných přípojných bodech.

TOPCShutdownConnectionPoint - tato třída není součástí ATL knihovny. Implementuje přípojný bod pro rozhraní IOPCShutdown. Je odvozena od ATL třídy **IConnectionPointImpl**. Má pouze jednu novou metodu pro vytváření události korespondující s voláním metody *ShutdownRequest* rozhraní IOPCShutdown.

Pro návrh a vytváření COM objektů existuje v prostředí C++ Builder zvláštní editor (Type Library Editor), který celý proces do značné míry automatizuje. Jeho použití je však určeno především pro vytváření nových COM objektů a není vhodný pro objekty, jejichž rozhraní jsou už předem definována, jako je tomu v případě OPC Serverů.

Registrace COM objektu v systému je zajištěna statickou funkcí *UpdateRegistry*, která je automaticky volána po spuštění aplikace s parametrem **/regserver**. Tato funkce využívá služeb dalšího z ATL objektů **TComServerRegistrarT**. Jedná se opět o šablonu, která při rozvíjení získá informace o příslušném COM objektu (CLSID, ProgID, popis), deklarované pomocí sady maker. Tyto informace jsou pak zapsány do databáze registrů a tím je provedena registrace objektu.

Po zdědění vlastností výše uvedených objektů bylo nutné pouze implementovat metody OPC rozhraní IOPC_Server, IOPCCommon a IOPCHDA_SyncRead. Všechna tato rozhraní jsou blíže popsána v kapitole 3. Detaily týkající se implementace jednotlivých metod je možné nalézt v přiloženém zdrojovém kódu aplikace.

5.2.3 Třída CoOPCHDABrowserImpl

Tato třída implementuje rozhraní COM objektu IOPCHDA_Browser určeného pro procházení adresového prostoru serveru. Jak bylo uvedeno výše, využívají objekty této třídy databázové spojení objektu TfrMain. Pro zajištění základních funkcí požadovaných COM jsou opět použity ATL objekty popsané v odstavci 5.2.2.

Bylo tedy nutné implementovat pouze čtyři metody rozhraní IOPCHDA_Browser. Tyto metody získávají informace o adresovém prostoru serveru z tabulek NODEDEFS a ITEMDEFS pomocí SQL příkazu SELECT.

Metoda *GetEnum* vrací seznam metod a položek definovaných v daném místě adresového prostoru. Protože tyto seznamy mohou být poměrně rozsáhlé, je pro jejich předání klientovi využíváno zvláštního rozhraní StringEnum. Jedná se o standardní rozhraní odvozené od IEnum, které umožňuje pohyb v seznamu a čtení jednotlivých položek. Pro implementaci tohoto rozhraní byl vytvořen zvláštní objekt, který je vytvářen v rámci volání metody *GetEnum*.

5.2.4 Konfigurace databáze

Tato část aplikace je určena pro konfiguraci adresového prostrou serveru. Její spuštění je možné, pouze pokud k serveru není připojen žádný klient. Tím je zajištěna ochrana proti případným kolizím způsobeným změnou databáze (ne však pro případ, kdy je jedna databáze využívána několika servery). Konfigurační formulář využívá sadu grafických VCL komponent spojených přes BDE rozhraní přímo s jednotlivými tabulkami v databázi. Zajišťuje tak automatickou změnu databáze při změnách údajů ve formuláři.

Adresový prostor je zobrazen ve stromové struktuře, do které lze přidávat nové uzly (Node) a položky (Item) jako následníky dříve definovaných uzlů. Pro každou položku je nutné

zadat alespoň její jméno (musí být unikátní v rámci databáze) a datový typ. Atributy položek lze vyplnit buď manuálně nebo, pro položky získávané z OPC DA serverů, automaticky volbou v adresovém prostoru daného serveru. Pro spojení s OPC DA servery a pro procházení jejich adresového prostoru je použit objekt TOPCServerConnection popsaný v odstavci 5.1.1. Atributy takto definovaných položek pak obsahují všechny potřebné informace pro jejich automatický záznam pomocí programu DB OPC Client.

Při odstraňování položek z adresového prostrou je možné odstranit z databáze i všechny záznamy s hodnotami dané položky.

Protože všechny prováděné změny jsou okamžitě zaznamenávány do databáze, projeví se ihned po ukončení konfigurace, bez nutnosti nového spouštění serveru.

5.3 Komponenta HDA Client

Prostředí Borland C++ Builder nabízí možnost vytváření nových VCL komponent a jejich následnou integraci do vytvářených aplikací. Komponenty jsou samostatné nezávislé objekty s definovanou funkcí (podobně jako COM objekty), z nichž lze sestavovat výslednou aplikaci v duchu objektového programování. Interakce s okolním světem je zajištěna pomocí vlastností (properties), metod a událostí (events). Více informací o vytváření VCL komponent je možné nalézt v [2].

Protože použití VCL komponent s sebou přináší celou řadu výhod a především usnadňuje tvorbu aplikací, rozhodl jsem se implementovat komunikační jádro HDA klienta v této podobě. Výsledná komponenta umožňuje spojení s jakýmkoliv serverem standardu OPC HDA (podobně jako objekty popsané v 5.1.1 pro rozhraní OPC DA) a využívání jeho služeb. Je možné získat seznam HDA serverů instalovaných na počítači specifikovaném jménem uzlu, navázat spojení se zvoleným serverem a využívat metody jeho rozhraní. Hodnoty položek a jejich atributů získaných ze serveru jsou uživateli přístupné pomocí vlastností. Každá komponenta umožňuje spojení s jedním serverem a čtení hodnot maximálně deseti položek z tohoto serveru. Pro každou položku lze číst hodnoty z databáze jednou z metod serveru popsaných v 3.3.4. Načtené hodnoty jsou pak přístupné pomocí vlastností komponenty. Dostupné jsou rovněž aktuální hodnoty všech atributů sledovaných položek. Po navázání spojení se serverem získá klient automaticky seznam definovaných agregačních funkcí a atributů spolu s jejich popisem.

Rozeberme nyní podrobněji vlastnosti a metody této komponenty, jejich účel a parametry.

tabulka 5-D Seznam vybraných vlastností VCL komponenty HDAClient

Jméno vlastnosti	**Popis**
`Names[i]`	Pole jmen definovaných položek. Pokud na pozici dané indexem `i` není položka definována, je vrácen prázdný řetězec.
`Values[i]`	Pole polí hodnot definovaných položek. Index `i` specifikuje položku. Hodnoty jsou předány pomocí objektu Variant obsahujícího pole hodnot datového typu určeného datovým typem položky.
`TimeStamps[i]`	Pole polí časových známek definovaných položek. Časové známky jsou opět předávány jako objekt Variant s polem záznamů typu TDateTime.
`Qualities[i]`	Pole polí kvalit definovaných položek. Kvality jsou uloženy jako Variant s polem typu DWORD.
`Attributes[i]`	Pole s hodnotami atributů pro každou položku specifikovanou indexem `i`. Atributy každé položky jsou umístěny v objektu Variant s polem záznamů typu String.
`Branches`	Seznam jmen větví (uzlů) v aktuálním místě adresového prostoru serveru.
`Leafs`	Seznam jmen listů (položek) v aktuálním místě adresového prostoru serveru.
`AggregatesNames`	Jména agregačních funkcí definovaných na serveru. Při použití metody ReadProcessed se agregační funkce specifikuje pomocí indexu v tomto seznamu.
`AggregatesDescriptions`	Popis a význam definovaných agregačních funkcí.
`AttributesNames`	Jména atributů definovaných na serveru. Záznamy v poli objektu Variant získaném pomocí Attributes[i] jsou řazeny ve stejném pořadí jako záznamy v tomto seznamu.
`AttributesDescriptions`	Popis a význam definovaných atributů.

tabulka 5-E Seznam vybraných metod VCL komponenty HDAClient

Deklarace metody	**Popis**
`Variant ListLocalServers (void)`	Vrací seznam ProgID OPC HDA serverů instalovaných na počítači.
`Variant ListRemoteServers (WideString NodeName)`	Vrací seznam ProgID OPC HDA serverů instalovaných na vzdáleném počítači, specifikovaném jménem uzlu.
`bool ConnectLocalServer (WideString ProgID)`	Naváže spojení s lokálním serverem specifikovaným pomocí jeho ProgID.
`bool ConnectRemoteServer (WideString ProgID, WideString NodeName)`	Naváže spojení se vzdáleným serverem specifikovaným pomocí jeho ProgID a jménem uzlu.
`bool BrowseUp (void)`	Změní aktuální pozici v adresovém prostoru na nadřazenou úroveň.
`bool BrowseDown (WideString NodeName)`	Změní aktuální pozici v adresovém prostoru na nižší úroveň specifikovanou jménem uzlu (větve).
`bool SetItemName (int Index, WideString ItemID)`	Nastaví jméno sledované položky na pozici danou indexem. Jména je možné získat z vlastnosti Leafs.
`bool ReadRawData (int Index, Variant Start, Variant End, int NumVal)`	Pro specifikovanou položku volá metodu *ReadRaw* rozhraní IOPCHDA_SyncRead. *Start* a *End* specifikují časový interval pomocí objektu TDateTime (nebo kompatibilního). *NumVal* udává maximální počet navrácených hodnot.
`bool ReadProcessedData (int Index, Variant Start, Variant End, double ResInt, int AgrInd)`	Pro specifikovanou položku volá metodu *ReadProcessed* rozhraní IOPCHDA_SyncRead. *ResInt* udává vzorkovací interval v sekundách. *AgrInd* specifikuje agregační funkci (indexem v rámci seznamu AggregatesNames).

`bool ReadDataAtTime (int Index, Variant TimeArray)`	Pro specifikovanou položku volá metodu *ReadAtTime* rozhraní IOPCHDA_SyncRead. Požadované časy jsou předány jako pole prvků TDateTime v objektu Variant.
`bool ReadAttributes (int Index)`	Pro specifikovanou položku přečte aktuální hodnoty všech jejích atributů.

Popsaná komponenta je navržena tak, aby co nejvíce usnadnila integrování OPC HDA do aplikací, ve kterých je používána, i za cenu jistých omezení (např. omezený počet sledovaných položek). Všechny datové typy jejích vlastností a parametrů metod jsou Automation Compatible, což velmi usnadňuje tvorbu následné ActiveX komponenty popsané blíže v kapitole 5.7.

5.4 HDA Client

Tento klient umožňuje grafickou prezentaci hodnot získaných z OPC HDA Serveru. Pro komunikaci se serverem používá služeb komponenty popsané v kapitole 5.3. Klient tvoří grafické rozhraní mezi touto komponentou a uživatelem, umožňující jednoduché spojení s jakýmkoliv dostupným serverem, prohlížení jeho adresového prostoru a čtení hodnot vybraných položek. Ovládání tohoto programu je blíže popsáno v příloze D.

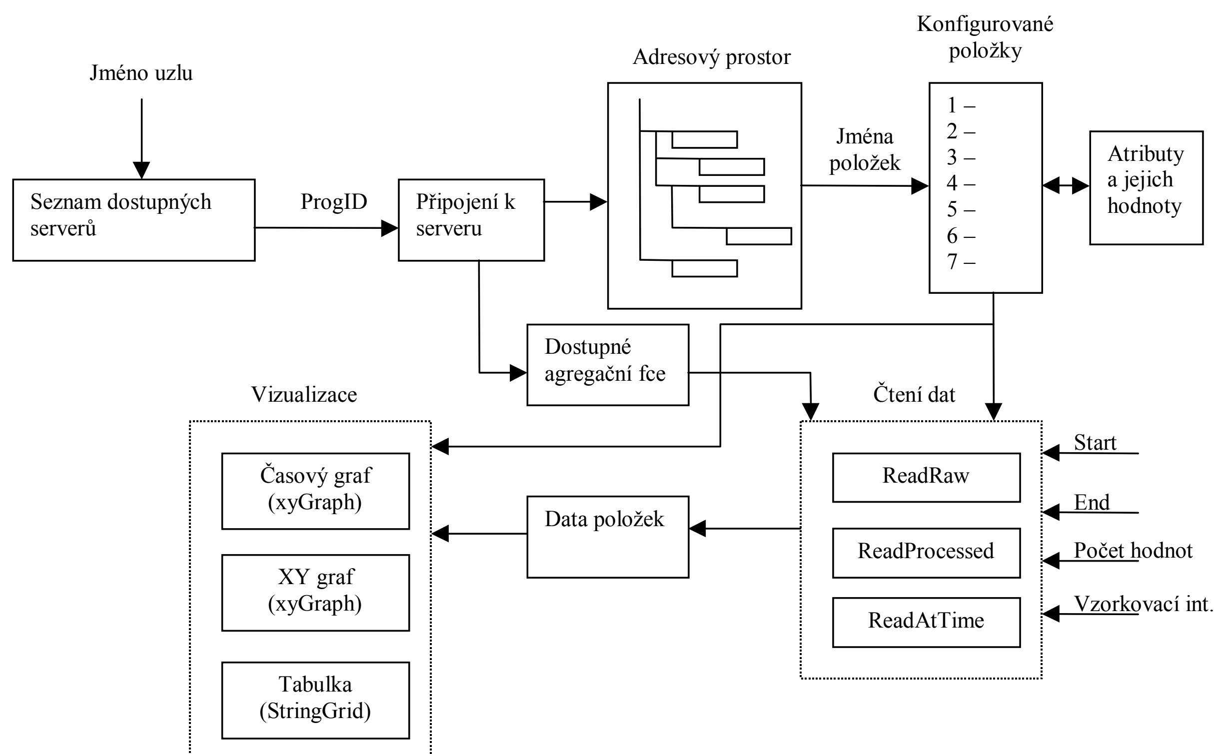
Protože celý program je postaven na komponentě popsané v kapitole 5.3, umožňuje současné zpracování hodnot maximálně deseti položek. Tyto položky mohou být vybrány z adresového prostoru připojeného serveru, zobrazeného v grafické podobě jako stromová struktura. Pro každou z vybraných položek je možné číst její hodnoty. Časové intervaly jsou zadávány pomocí editačních oken vybavených zvláštní maskou pro tento účel. O veškerých prováděných operacích a jejich úspěšnosti jsou udržovány záznamy.

Načtené hodnoty lze zobrazit třemi způsoby. Prvním způsobem je časový graf. Pro každý zobrazený graf je vytvořeno zvláštní okno a lze tedy zvlášť nastavovat vlastnosti každého grafu. V jednom grafu lze zobrazit libovolný počet položek. Pokud je zvolena vlastnost dynamický graf, jsou průběhy při načtení nových hodnot automaticky aktualizovány. Grafy je možné ukládat do souboru jako obrázky, nebo tisknout na tiskárně. Pro vytvoření grafů byla použita komponenta xyGraph dosažitelná z [14].

Další možností pro zobrazení je použití XY grafu, kdy pro záznamy na ose X jsou použity hodnoty zvolené z položek. Na ose Y jsou pak vynášeny hodnoty libovolného počtu dalších položek. Pole hodnot pro všechny zobrazované položky musí mít samozřejmě stejný počet záznamů. Grafy jsou opět vytvořeny s použitím komponenty xyGraph.

Posledním způsobem vizualizace dat je jejich zobrazení v podobě tabulky. Zobrazeny jsou jak hodnoty (v textové podobě), tak i příslušné časové známky s rozlišením na milisekundy a kvality převedené do slovního vyjádření. V jedné tabulce lze zobrazit hodnoty více položek. Sloupce tabulky pak lze různým způsobem přerovnávat a rušit.

Struktura celé klientské aplikace je nejlépe patrná z obr. 5.2.



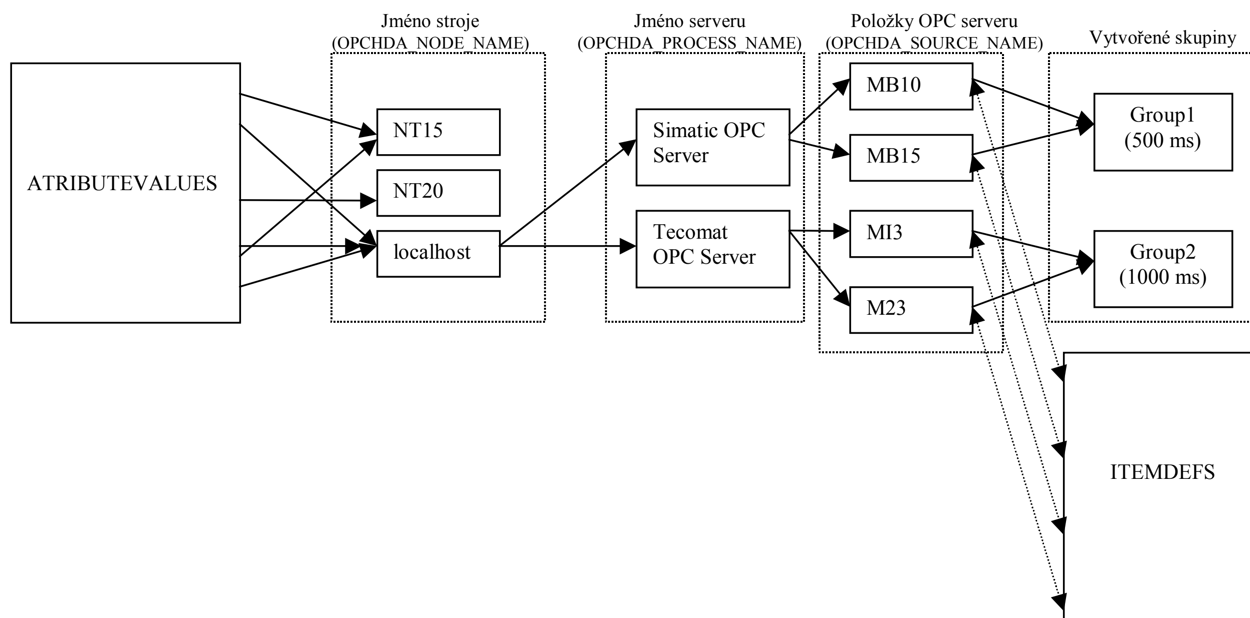
obr. 5.2 Struktura a schéma funkce aplikace HDA Client

5.5 DB OPC Client

Velmi důležitou součástí systému pro práci s historickými daty je nástroj umožňující sběr těchto dat z technologie a jejich záznam do databáze. Pro navržený HDA server jsem implementoval dva takové nástroje. První z nich, kterým se budeme zabývat v této kapitole, je specielní klient pro rozhraní OPC DA.

Tento klient umožňuje připojení k jakémukoliv OPC DA 2.0 serveru, periodický sběr hodnot vybraných položek z tohoto serveru a jejich záznam do databáze. Každá položka HDA serveru má kromě svých hodnot také množství atributů nesoucích doplňkové informace o položce. Tyto atributy je možné specifikovat například při vytváření této položky v adresovém prostoru pomocí nástroje DB Config tak, jak je popsáno v odstavci 5.2.4. Ve specifikacích OPC je popsána sada standardních atributů, z nichž čtyři poskytují informace o zdroji uvedené položky (jejich bližší popis lze nalézt v příloze B). S využitím těchto čtyřech atributů je možné

provést jednoznačné mapování dané položky OPC HDA Serveru na určitou položku OPC DA Serveru spolu s veškerými informacemi o umístění DA serveru, čehož je využito v programu DB OPC Klient.



obr. 5.3 Příklad vytváření skupin položek získávaných z OPC DA Serverů

Klient se pomocí standardního postupu, popsaného v 4.1.1, připojí k databázi, do které mají být zaznamenávána data. Z tabulek ITEMDEFS a ATTRIBUTEVALUES pak získá nejprve seznam jmen všech počítačů, na kterých jsou umístěny OPC servery sloužící jako zdroj pro definované položky. Pro každý počítač je získán seznam OPC serverů a seznam položek získávaných z jednotlivých OPC serverů. Uživatel má možnost procházet tyto seznamy a sdružovat položky náležející k jednomu serveru do skupin s definovanou frekvencí aktualizace hodnot. Výsledkem je sada skupin položek, přičemž v každé skupině jsou pouze položky náležející k jednomu serveru.

Pro vybrané skupiny lze spustit záznam dat jejich položek. Pomocí objektů popsaných v kapitole 5.1.1 je navázáno spojení s příslušnými servery a na nich vytvořeny odpovídající OPC Skupiny. Servery pak samy zajistí volání metody *OnDataChange* podle zvolené frekvence aktualizace skupiny. Při každém zavolání této metody jsou nové hodnoty změněných položek zaznamenány do databáze. Záznam hodnot je ukončen přerušením spojení mezi klientem a OPC Servery.

5.6 DB Uni Client

Dalším nástrojem pro záznam dat do databáze je klient pro komunikační server UniServer vytvořený v rámci diplomové práce popsané v [9]. Jedná se o univerzální komunikační program využívající protokol TCP/IP pro přenos dat. Pro tento server byla vytvořena celá řada klientů umožňujících získávání dat z nejrůznějších aplikací (např. Matlab, Excel, sítě MODBUS TCP, aplikací podporujících DDE). Každý klient tohoto systému má přístup k datům poskytovaným všemi ostatními klienty a tak je možné s pomocí tohoto nástroje sbírat data z širokého spektra aplikací.

Struktura aplikace je velmi podobná aplikaci popsané v kapitole 5.5. Pro identifikaci zdroje dat pro položky jsou opět použity její atributy. Atribut OPCHDA_PROCESS_NAME specifikuje síťové jméno klienta, který je zdrojem pro danou položku. V rámci tohoto klienta je položka identifikována pomocí atributu OPCHDA_SOURCE_NAME. Určení jména stroje, na kterém se počítač nachází, není v tomto případě potřebné, protože klient je jednoznačně identifikován svým jménem.

5.7 ActiveX HDA Client

Aby bylo možné data dostupná z OPC HDA serverů zpracovávat v dalších aplikacích, byla vytvořena ActiveX komponenta, umožňující spojení s těmito servery. Je založena na VCL komponentě popsané v kapitole 5.3. Protože všechny metody a vlastnosti vzorové komponenty pracují pouze s Automation Compatible typy, bylo možné použít automatický nástroj pro generování ActiveX objektů, který je součástí prostředí Borland C++ Builder. Vygenerovaná komponenta obsahuje, až na drobné výjimky, stejné metody a vlastnosti jako zdrojový objekt. ActiveX specifikace nepodporuje vlastnosti typu pole a proto je nutné přistupovat k datům těchto vlastností pomocí zvláštních metod.

Výslednou komponentu je možné použít v kterékoliv aplikaci podporující technologii ActiveX (např. Webové prohlížeče, programy MS Office, Matlab a mnoho dalších). Pro zjednodušení a urychlení použití byl ke komponentě přidán ovládací formulář totožný s formulářem programu HDA Client. Tak je možné s minimálním množstvím programového kódu integrovat komponentu do aplikace, navázat spojení s HDA serverem a načíst data pro žádané položky. Další zpracování těchto dat lze pak provádět v hostitelské aplikaci. Příklad použití této komponenty v aplikaci MS Excel je na přiloženém CD.

6 Závěr

Ve své diplomové práci jsem navrhl a implementoval server pro zpracování historických dat v souladu se specifikací OPC Historical Data Access 1.0. OPC specifikace jsou rozšířenými a uznávanými standardy pro výměnu dat v oblasti řídicí techniky. Vytvořený server zpřístupňuje klientům data uložená v navržené databázi. Čtení a zpracování dat je možné pomocí všech metod definovaných v OPC HDA specifikaci pro rozhraní IOPCHDASync_Read. Ostatní volitelná rozhraní pro práci s historickými daty nejsou implementována.

Použitá databáze procesních dat byla vytvořena pod databázovým serverem MySQL, který patří k nejrozšířenějším SQL serverům a je šířen jako Open Source pod GPL licencí. Ke spojení Historical Data Access a databázového serveru je využito ODBC rozhraní. Toto rozhraní je součástí všech systémů Windows, a proto může být pro uložení databáze procesních dat použito jakéhokoliv databázového serveru podporujícího ODBC.

V databázi jsou uložena jak procesní data, tak i informace o konfiguraci adresového prostoru serveru a o definovaných atributech a agregačních funkcích. Agregační funkce jsou definovány přímo v databázi pomocí SQL příkazů s parametry, jež jsou při použití funkce nahrazeny skutečnými hodnotami. Tento přístup umožňuje snadné rozšíření funkcí serveru a konfiguraci adresového prostoru. Konfigurační nástroj je součástí programu serveru.

Pro vizualizaci historických dat jsem vytvořil klient spolupracující s jakýmkoliv OPC HDA serverem. Klient umožňuje současně pracovat maximálně s deseti položkami. Data mohou být čtena a zpracovávána všemi metodami rozhraní IOPCHDASync_Read. Získaná data jsou graficky interpretována pomocí tabulek a časových nebo XY grafů. Jádro klienta, zajišťující komunikaci s HDA serverem, je implementováno jako VCL komponenta, snadno použitelná v dalších aplikacích. Celý klient, bez vizualizačních prostředků, je rovněž vytvořen v podobě ActiveX objektu, který umožňuje přenos historických dat do všech aplikací podporujících ActiveX technologii.

Sběr technologických dat do databáze je zajištěn dvěma specializovanými klienty. První používá pro získávání dat velmi rozšířeného rozhraní OPC Data Access a tak umožňuje přístup k datům z většiny dostupných zařízení. Druhý databázový klient spolupracuje s TCP/IP serverem UniServer vytvořeným v rámci [9]. Pro tento server byla implementována sada klientů zpřístupňujících data z řady různých aplikací.

Všechny aplikace byly vyvinuty v prostředí Borland C++ Builder 5.0 s použitím integrovaných nástrojů a knihoven VCL. Pro implementaci COM objektů jsem využil knihoven ATL dodávaných spolu s vývojovým prostředím. Spojení mezi aplikací serveru a databázovým

rozhraním ODBC je realizováno pomocí Borland Database Engine (BDE) a příslušných databázových komponent.

V teoretické části práce jsem se zaměřil především na popis použitých technologií. Je popsána technologie COM s ohledem na vytváření objektů a rozhraní podle specifikací OPC. Dále jsou popsány použité OPC specifikace, jejich účel a struktury typických aplikací. Zvláštní kapitola je věnována serveru MySQL, připojení k databázovému serveru, návrhu databáze a její struktuře.

Možné rozšíření vytvořeného serveru je dáno samotnou OPC specifikací. Ta zahrnuje popis řady volitelných rozhraní pro výměnu historických dat, která nebyla implementována. Pro zvýšení efektivnosti čtení dat by bylo vhodné implementovat asynchronní rozhraní IOPCHDAAsync_Read. Specifikace rovněž popisuje dvě rozhraní pro zápis dat do databáze. Standardizace vkládání dat by umožnila použít pro sběr dat klienty různých dodavatelů. Užitečné by bylo také upravení definice a výpočtu agregačních funkcí tak, aby bylo možné použít externích výpočetních nástrojů pro složitější funkce. Pozornost rovněž zasluhuje způsob práce s časovými značkami s ohledem na přechod mezi letním a zimním časem.

Všechny vytvořené aplikace byly úspěšně testovány ve spojení se simulačními OPC DA servery firem Matrikon a Softing jako zdroji dat. Část dat byla rovněž uměle naplněna do databáze z jiných zdrojů. Testovací databáze obsahovala celkem 368013 záznamů. Ze serveru může být v rámci jednoho volání procedury načteno maximálně 1000 hodnot s příslušnými kvalitami a časovými známkami.

Hlavní výhodou serveru je jeho nenáročnost na cílový systém a univerzálnost díky zvoleným způsobům připojení k databázi a sběru dat. Jeho použití je možné ve všech 32 bitových verzích operačního systému Windows. Je vhodný spíše pro menší aplikace vyžadující současné ukládání maximálně několika desítek položek s periodou záznamu větší než 100 ms.

Chtěl bych poděkovat za pomoc pracovníkům oddělení řídicích systémů katedry řídicí techniky FEL ČVUT v Praze a svým kolegům studentům. Zvláště bych chtěl poděkovat Ing. Pavlu Burgetovi a Ing. Tomáši Novákovi za věcné připomínky a praktickou pomoc při realizaci diplomové práce.

7 Seznam použité literatury

[1] Borland C++ Builder 5.0 Help, Developing COM-based Applicaitons, Borland, 2000

[2] Borland C++ Builder 5.0 Help, Creating Custom Components, Borland, 2000

[3] Radek Mastný, Technologie COM a OPC, Diplomová práce, ČVUT v Praze, 2000

[4] OPC Data Access Custom Interface Standard 2.05, OPC Foundation, 2001

[5] OPC Overview 1.0, OPC Foundation, 1998

[6] OPC Common Definition and Interfaces 1.0, OPC Foundation, 1998

[7] OPC Historical Data Access Custom Interface Standard 1.1, OPC Foundation, 2001

[8] Jaroslav Pokorný,Ivan Halaška, Databázové systémy, ČVUT v Praze, 1999

[9] Michal Houdek, Sběr a zpracování dat průmyslových aplikací, Diplomová práce, ČVUT v Praze, 2003

[10] Microsoft SDK Help, MIDL Programmer's Reference, Microsoft, 1996

Internet:

[11] msdn.microsoft.com Microsoft Software Developer Network

[12] www.opcfoundation.org Organizace OPC Foundation

[13] www.mysql.com Firemní stránky MySQL AB

[14] kesral.com.au/xygraph Komponenta xyGraph

[15] www.opcconnect.com OPC Programmer's Connection

[16] www.matrikon.com Firemní stránky Matrikon

[17] www.softing.com Firemní stránky Softing

8 Přílohy

Příloha A - SQL definice tabulek

SQL příkazy pomocí kterých lze vytvořit databázi pro OPC HDA Server:

```
CREATE TABLE ITEMDEFS (hServer integer(10) not null, ItemID varchar(50) not null, NodeID integer(10) not null, DataType integer(10) not null, PRIMARY KEY (hServer))

CREATE TABLE ITEMVALUES (hServer integer(10) not null, Value varchar(250) not null, TimeStamp BIGINT(32) not null, Quality integer(10) not null, PRIMARY KEY (hServer, TimeStamp))

CREATE TABLE ATTRIBUTEDEFS (AttributeID integer(10) not null, AttributeName varchar(50) not null, DataType integer(10) not null, Description varchar(200), PRIMARY KEY (AttributeID))

CREATE TABLE ATTRIBUTEVAL (AttributeID integer(10) not null, hServer integer(10) not null, Value varchar(200), TimeStamp BIGINT(32), PRIMARY KEY (hServer, AttributeID))

CREATE TABLE CLIENTHANDLES (hServer integer(10) not null, ClientID integer(10) not null, hClient integer(10), PRIMARY KEY (ClientID, hServer))

CREATE TABLE AGGREGATEDEFS (AggregateID integer(10) not null, AggregateName varchar(50) not null, Description varchar(200), Equation varchar(250), ReturnType integer(10), PRIMARY KEY (AggregateID))

CREATE TABLE NODEDEFS (NodeID integer(10) not null, NodeName varchar(50) not null, ParentNodeID integer(10) not null, PRIMARY KEY (NodeID))

CREATE TABLE DATATYPES (DataType integer(10) not null, Description varchar(50), PRIMARY KEY (DataType))
```

Příloha B - Definované atributy položek

Všechny uvedené atributy jsou definovány v rámci OPC HDA specifikace.

ID	Jméno	Typ	Popis
1	OPCHDA_DATA_TYPE	integer	Specifikuje datový typ položky
2	OPCHDA_DESCRIPTION	string	Popisuje význam položky
7	OPCHDA_NODE_NAME	string	Určuje jméno stroje, na kterém je umístěn zdroj položky. Pro položky získávané z OPC je to jméno uzlu nebo IP adresa počítače, na kterém je umístěn server.
8	OPCHDA_PROCESS_NAME	string	Určuje jméno procesu (aplikace), která je zdrojem položky. Pro OPC položky je to ProgID OPC serveru.
9	OPCHDA_SOURCE_NAME	string	Jednoznačně identifikuje položku v rámci zdrojové aplikace. Pro OPC položky je to identifikátor (ItemID) položky v OPC serveru.
10	OPCHDA_SOURCE_TYPE	string	Určuje typ procesu, ze kterého je položka získávána (např. OPC).

Příloha C - Definované agregační funkce

Všechny uvedené agregační funkce jsou definovány v rámci OPC HDA specifikace.

ID	Jméno	Popis
1	OPCHDA_INTERPOLATIVE	Neprovádí výpočet agregační funkce.
2	OPCHDA_TOTAL	Vypočte časový integrál hodnot ve vzorkovacím rozsahu.
3	OPCHDA_AVERAGE	Vrací průměrnou hodnotu ve vzorkovacím intervalu.
5	OPCHDA_COUNT	Vrací počet zaznamenaných hodnot ve vzorkovacím intervalu.
8	OPCHDA_MINIMUM	Vrací minimální hodnotu ve vzorkovacím intervalu.
10	OPCHDA_MAXIMUM	Vrací maximální hodnotu ve vzorkovacím intervalu.
4	OPCHDA_TIMEAVERAGE	Vrací časově vážený průměr ve vzorkovacím intervalu.
7	OPCHDA_MINIMUMACTUALTIME	Vrací minimální hodnotu ve vzorkovacím intervalu a časovou známku této hodnoty.
9	OPCHDA_MAXIMUMACTUALTIME	Vrací maximální hodnotu ve vzorkovacím intervalu a časovou známku této hodnoty.
11	OPCHDA_START	Vrací první hodnotu ve vzorkovacím intervalu a její časovou známku.
12	OPCHDA_END	Vrací poslední hodnotu ve vzorkovacím intervalu a její časovou známku.
18	OPCHDA_RANGE	Vrací rozdíl mezi min a max hodnotou ve vzork. int.

ID	SQL příkaz
1	SELECT Value as Value, MIN(TimeStamp) as TimeStamp, Quality as Quality FROM itemvalues WHERE (hServer = :hServer) AND (TimeStamp >= :Start) AND (TimeStamp <= :End) GROUP BY (FLOOR((TimeStamp - :Start)/:Resample)) LIMIT :Limit
2	SELECT SUM(Value) as Value, MIN(TimeStamp) as TimeStamp, Quality as Quality FROM itemvalues WHERE (hServer = :hServer) AND (TimeStamp >= :Start) AND (TimeStamp <= :End) GROUP BY (FLOOR((TimeStamp - :Start)/:Resample)) LIMIT :Limit
3	SELECT AVG(Value) as Value, MIN(TimeStamp) as TimeStamp, Quality as Quality FROM itemvalues WHERE (hServer = :hServer) AND (TimeStamp >= :Start) AND (TimeStamp <= :End) GROUP BY (FLOOR((TimeStamp - :Start)/:Resample)) LIMIT :Limit
5	SELECT COUNT(Value) as Value, MIN(TimeStamp) as TimeStamp, Quality as Quality FROM itemvalues WHERE (hServer = :hServer) AND (TimeStamp >= :Start) AND (TimeStamp <= :End) GROUP BY (FLOOR((TimeStamp - :Start)/:Resample)) LIMIT :Limit
8	SELECT MIN(Value) as Value, MIN(TimeStamp) as TimeStamp, Quality as Quality FROM itemvalues WHERE (hServer = :hServer) AND (TimeStamp >= :Start) AND (TimeStamp <= :End) GROUP BY (FLOOR((TimeStamp - :Start)/:Resample)) LIMIT :Limit
10	SELECT MAX(Value) as Value, MIN(TimeStamp) as TimeStamp, Quality as Quality FROM itemvalues WHERE (hServer = :hServer) AND (TimeStamp >= :Start) AND (TimeStamp <= :End) GROUP BY (FLOOR((TimeStamp - :Start)/:Resample)) LIMIT :Limit
4	SELECT AVG(Value)/(:Resample/10000000) as Value,MIN(TimeStamp) as TimeStamp, Quality as Quality FROM itemvalues WHERE (hServer = :hServer) AND (TimeStamp >= :Start) AND (TimeStamp <= :End) GROUP BY (FLOOR((TimeStamp - :Start)/:Resample)) LIMIT :Limit

7	SELECT MIN(Value) as Value, TimeStamp as TimeStamp, Quality as Quality FROM itemvalues WHERE (hServer = :hServer) AND (TimeStamp >= :Start) AND (TimeStamp <= :End) GROUP BY (FLOOR((TimeStamp - :Start)/:Resample)) LIMIT :Limit
9	SELECT MAX(Value) as Value, TimeStamp as TimeStamp, Quality as Quality FROM itemvalues WHERE (hServer = :hServer) AND (TimeStamp >= :Start) AND (TimeStamp <= :End) GROUP BY (FLOOR((TimeStamp - :Start)/:Resample)) LIMIT :Limit
11	SELECT Value as Value, MIN(TimeStamp) as TimeStamp, Quality as Quality FROM itemvalues WHERE (hServer = :hServer) AND (TimeStamp >= :Start) AND (TimeStamp <= :End) GROUP BY (FLOOR((TimeStamp - :Start)/:Resample)) LIMIT :Limit
12	SELECT Value as Value, MAX(TimeStamp) as TimeStamp, Quality as Quality FROM itemvalues WHERE (hServer = :hServer) AND (TimeStamp >= :Start) AND (TimeStamp <= :End) GROUP BY (FLOOR((TimeStamp - :Start)/:Resample)) LIMIT :Limit
18	SELECT (MAX(Value)-MIN(Value)) as Value, MIN(TimeStamp) as TimeStamp, Quality as Quality FROM itemvalues WHERE (hServer = :hServer) AND (TimeStamp >= :Start) AND (TimeStamp <= :End) GROUP BY (FLOOR((TimeStamp - :Start)/:Resample)) LIMIT :Limit

Příloha D - Instalace a popis ovládání vytvořených programů

Instalace programů

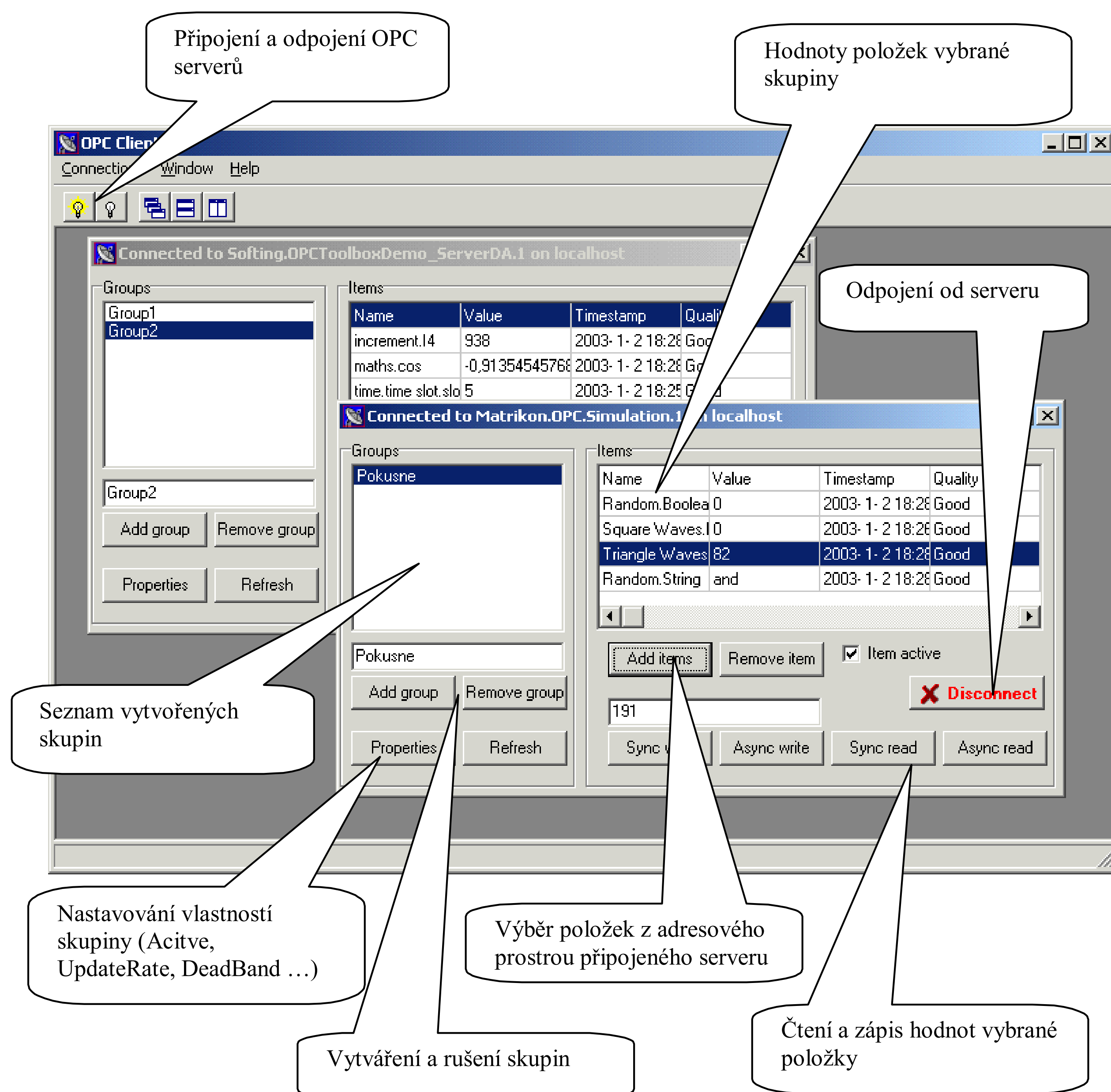
Pomocí aplikace Install Shield 2.0, dodávané společně s prostředím C++ Builder, jsem vytvořil společný instalační program pro všechny vytvořené aplikace, obsažený na přiloženém CD-ROM. Program umožňuje manuální volbu, které aplikace mají být instalovány, nebo typickou a minimální instalaci. Kromě vytvořených aplikací je možné instalovat dvě databáze pro OPC HDA Server. Testovací databáze obsahuje definici několika položek a záznamy testovacích dat pro tyto položky. Standardní databáze pak neobsahuje žádné položky, pouze potřebné definice atributů, agregačních funkcí a datových typů.

Instalační program rovněž zajistí registraci potřebných OPC proxy/stub knihoven (pokud již nejsou instalovány) a automatickou registraci všech souborů obsahujících COM objekty.

V případě instalace databáze jsou automaticky vytvořeny příslušné ODBC aliasy a instalováno jádro BDE, nutné pro spojení aplikací s databázovým serverem.

Veškeré změny provedené na cílovém systému jsou zaznamenávány, takže je v případě potřeby možné odinstalování celého balíku.

Ovládání programu OPC Client



Ovládání programu HDA Server a DB Config

Povolení záznamu událostí do souboru

Dostupná jména ODBC aliasů pro instalované databáze

Maximální velikost log souboru (nejstarší záznamy jsou odstraňovány)

Adresář, ve kterém je vytvořen log soubor (TSHDASrv.log)

Jméno aktuálně používaného ODBC aliasu (změna se projeví až po restartu serveru)

Uložení konfigurace do ini souboru v systémovém adresáři

Odpojení všech klientů (volání události ShutdownRequest) a ukončení činnosti serveru

Minimalizace okna serveru a umístění ikony do icon tray

Spuštění programu pro konfiguraci databáze (pouze pokud není aktivní žádné připojení)

Aktuální struktura adresového prostrou

Jméno a rodič vybraného uzlu

Záznam změn do databáze

Hodnoty definovaných atributů

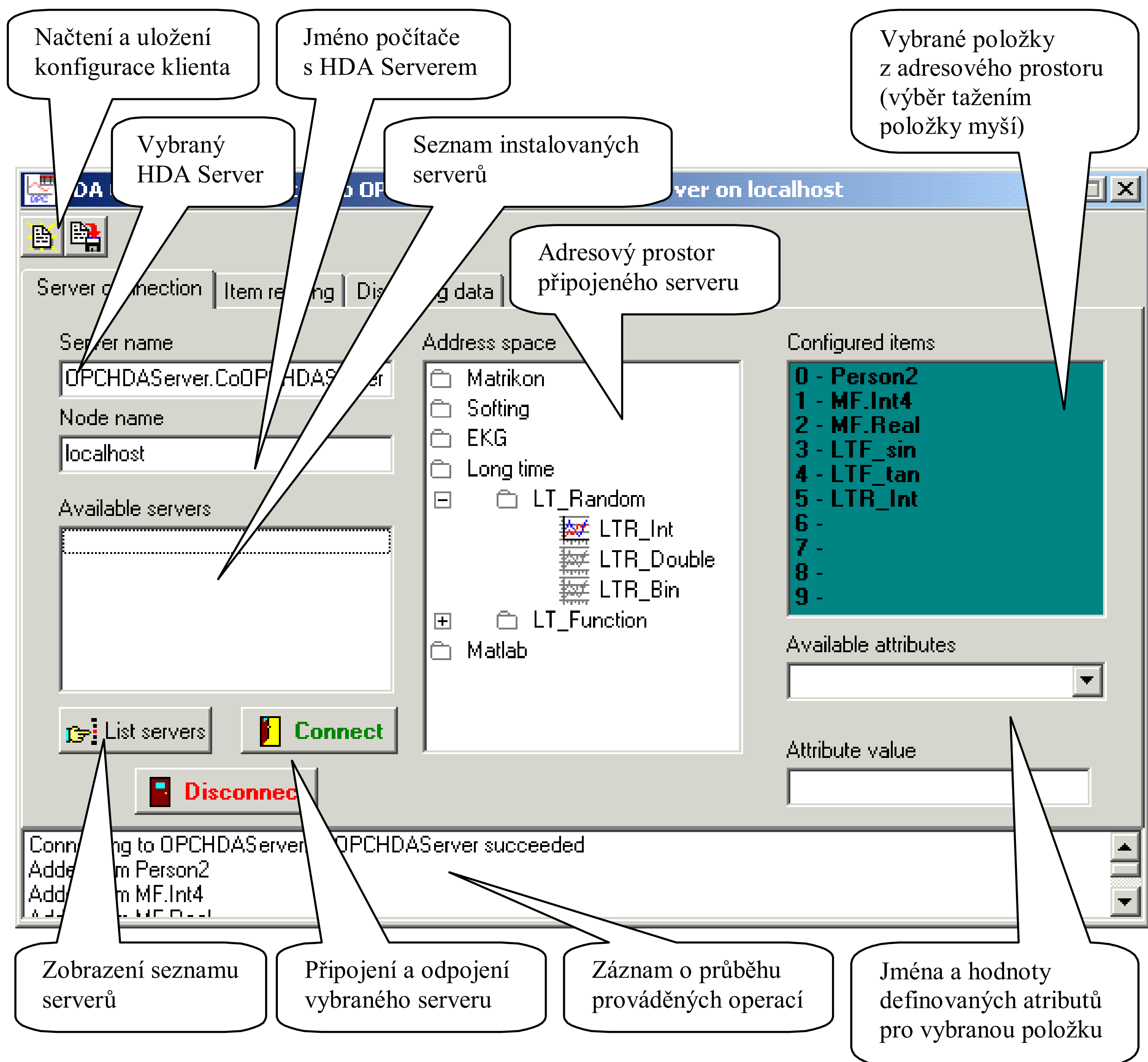
Občerstvení adresového stromu

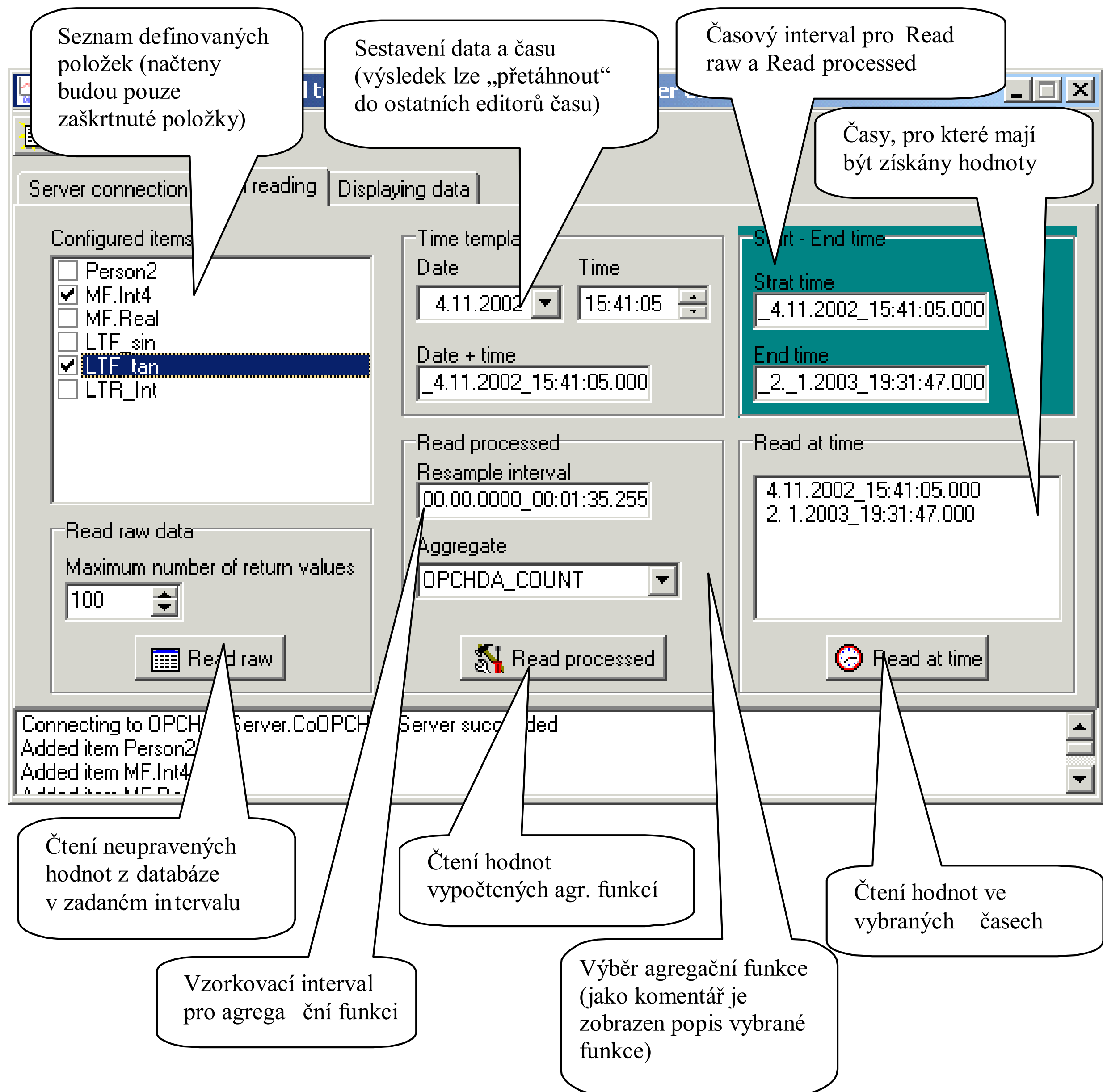
Jméno, datový typ a rodič vybrané položky

Jméno a hodnota vybraného atributu

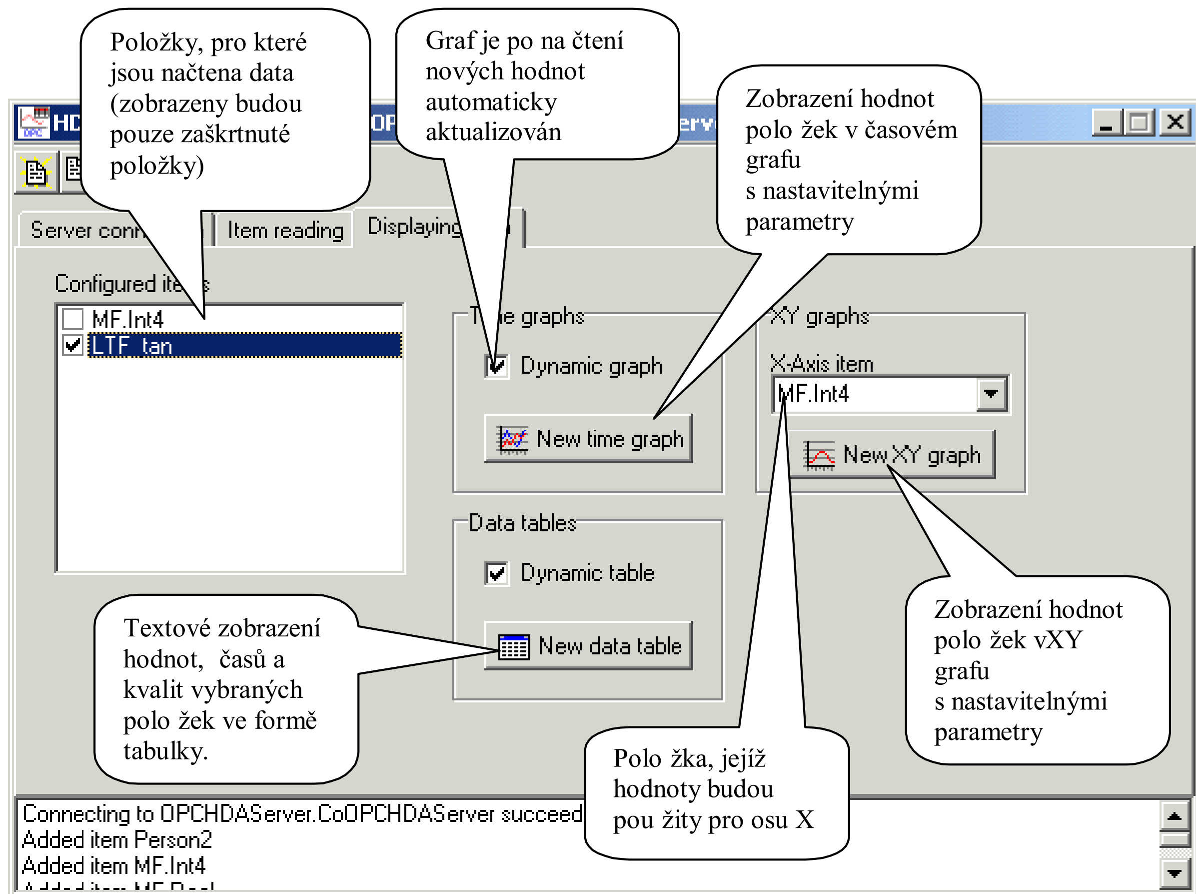
Definice vybrané položky z OPC DA Serveru

Ovládání programu HDA Client

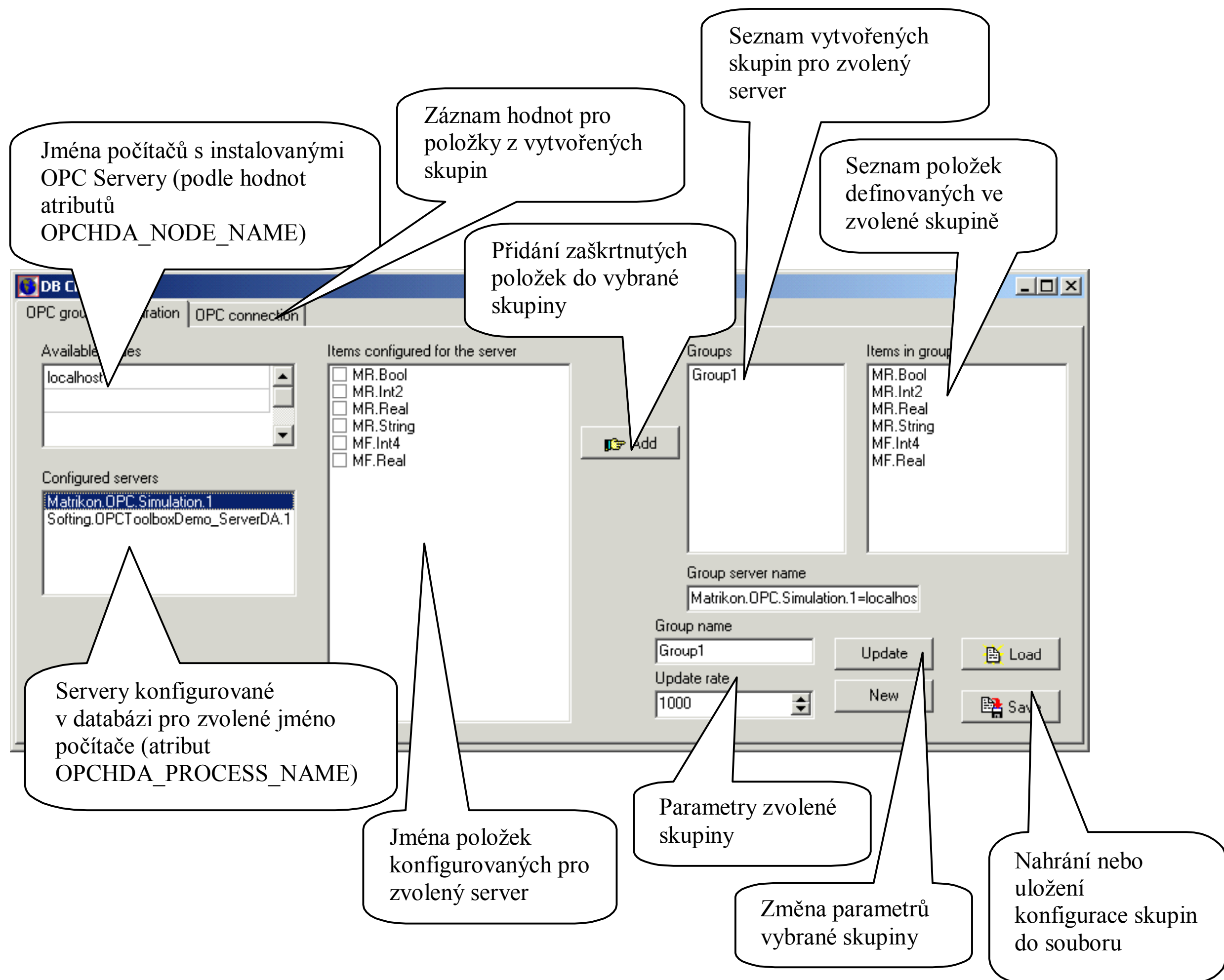




Seznam definovaných položek (načteny budou pouze zaškrtnuté položky)
Sestavení data a času (výsledek lze „přetáhnout“ do ostatních editorů času)
Časový interval pro Read raw a Read processed
Časy, pro které mají být získány hodnoty
Server connection
reading
Displaying data
Configured items
Person2
MF.Int4
MF.Real
LTF_sin
LTF_tan
LTR_Int
Time templa
Date
Time
4.11.2002
15:41:05
Date + time
_4.11.2002_15:41:05.000
Start - End time
Strat time
_4.11.2002_15:41:05.000
End time
_2._1.2003_19:31:47.000
Read processed
Resample interval
00.00.0000_00:01:35.255
Aggregate
OPCHDA_COUNT
Read at time
4.11.2002_15:41:05.000
2. 1.2003_19:31:47.000
Read raw data
Maximum number of return values
100
Read raw
Read processed
Read at time
Connecting to OPCH
Server.CoOPCH
Server succ
ded
Added item Person2
Added item MF.Int4
Čtení neupravených hodnot z databáze v zadaném intervalu
Čtení hodnot vypočtených agr. funkcí
Čtení hodnot ve vybraných časech
Vzorkovací interval pro agrega ční funkci
Výběr agregační funkce (jako komentář je zobrazen popis vybrané funkce)


Položky, pro které jsou načtena data (zobrazeny budou pouze zaškrtnuté položky)
Graf je po na čtení nových hodnot automaticky aktualizován
Zobrazení hodnot polo žek v časovém grafu s nastavitelnými parametry
Server conn
Item reading
Displaying
Configured ite
MF.Int4
LTF_tan
Time graphs
Dynamic graph
New time graph
XY graphs
X-Axis item
MF.Int4
New XY graph
Data tables
Dynamic table
New data table
Zobrazení hodnot polo žek vXY grafu s nastavitelnými parametry
Textové zobrazení hodnot, časů a kvalit vybraných polo žek ve formě tabulky.
Polo žka, jejíž hodnoty budou pou žity pro osu X
Connecting to OPCHDAServer.CoOPCHDAServer succeed
Added item Person2
Added item MF.Int4

Ovládání programu DB OPC Client

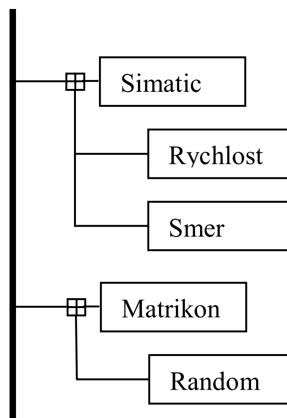


Příloha E - WWW servery související s uvedenou tématikou

- www.opcfoundation.org – oficiální stránky organizace OPC Foundation. Zde je možné nalézt texty všech OPC specifikací. Pro každou specifikaci jsou rovněž k dispozici popisy použitých COM rozhraní v jazyce IDL a příslušné proxy/stub knihovny. Stránky obsahují databázi všech registrovaných dodavatelů OPC kompatibilního software a jejich produktů.
- www.mysql.com – stránky společnosti MySQL AB zabývající se vývojem databázového serveru MySQL. K dispozici jsou všechny verze serveru, spolu s manuály a dalšími podpůrnými nástroji a ovladači. Pro každou aplikaci je možné získat rovněž zdrojové kódy.
- www.matrikon.com – firma Matrikon se zabývá vývojem softwaru pro řídicí techniku. Je jedním ze zakládajících členů OPC Foundation. Na jejích stránkách jsou k dispozici volně ke stažení různé druhy OPC Serverů a dalších užitečných nástrojů pro práci s OPC. Jedná se o profesionální nástroje, velmi vhodné jak pro testovací účely, tak i pro použití v průmyslu.
- www.softing.com – firma Softing je rovněž členem OPC Foundation. Na jejích stránkách je k dispozici velmi užitečný testovací OPC DA server.
- www.opcconnect.com – stránky OPC Programers Connection jsou určeny pro komunikaci mezi vývojáři OPC Aplikací. Část stránek je zaměřena speciálně na vývoj OPC aplikací v programovacích prostředcích Borland C++ Builder a Borland Delphi. K dispozici je mnoho ukázek zdrojových kódů a komponent pro tato prostředí a programovací jazyky.
- msdn.microsoft.com/library/us/dnguion/html/msdn_drguion020298.asp – popis základů programování technologie COM od firmy Microsoft. Popis je psán velmi názornou a přístupnou formou, včetně ukázek zdrojových kódů v jazyce C.

Příloha F – Ukázka definice adresového prostoru serveru

Jako příklad uvažujme server který má definovanou stromovou adresovou strukturu obsahující dva uzly a tři položky podle následujícího obrázku:



Předpokládejme, že data pro všechny definované položky jsou získávána z příslušných OPC DA serverů umístěných na vzdálených počítačích. Nyní uvedeme jak by takováto struktura byla definována v databázi.

ITEMDEFS			
hServer	**ItemID**	**NodeID**	**DataType**
1	Rychlost	2	5
2	Smer	2	2
3	Random	3	5

NODEDEFS		
NodeID	**NodeName**	**ParentNode**
1	Root	0
2	Simatic	1
3	Matrikon	1

ATTRIBUTEVAL			
AttributeID	**hServer**	**Value**	**TimeStamp**
7	1	NT15	3.1.2003 14:45:37.002
8	1	OPC.SimaticNET	3.1.2003 14:45:37.002
9	1	Memory.MD5	3.1.2003 14:45:37.002
10	1	OPC	3.1.2003 14:45:37.002
7	2	NT15	3.1.2003 17:38:12.123
8	2	OPC.SimaticNET	3.1.2003 17:38:12.123
9	2	Memory.MX12	3.1.2003 17:38:12.123
10	2	OPC	3.1.2003 17:38:12.123
7	3	NT08	2.2.2003 12:15:06.452
8	3	Matrikon.OPC.Simul	2.2.2003 12:15:06.452
9	3	Random.Double	2.2.2003 12:15:06.452
10	3	OPC	2.2.2003 12:15:06.452