České vysoké učení technické v Praze
Fakulta elektrotechnická

DIPLOMOVÁ PRÁCE

OPC Server

Praha, 2006 **Autor: Ondřej Šťavík**

Prohlášení

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

V Praze dne _____ _____

podpis

Poděkování

Děkuji především vedoucímu diplomové práce Ing. Richardu Šustovi, PhD. za pomoc a cenné připomínky.

Abstrakt

Tato diplomová práce se zabývá využitím specifikace OPC Data Access k přístupu na fyzické zařízení. Hlavní náplní je vývoj OPC serveru pro I/O kartu Advantech PCI 1750. Server podporuje povinné části OPC Data Access 2.05. Z volitelných částí je implementováno rozhraní IOPCBrowseAddressSpace. K demonstraci funkcí serveru byl vytvořen i jednoduchý OPC klient.

Dalším úkolem této práce je vývoj konfiguračního prostředí (manažera) k serveru. Ten má za úkol zpřístupnit uživateli některá globální nastavení serveru, nedostupná z OPC klientů.

V teoretické části práce je stručný úvod do některých typů meziprocesové komunikace ve Windows. Od těch jednodušších až po model COM, který je základem technologie OPC. Dále se teoretická část zabývá popisem specifikace OPC Data Access. Praktická část se pak věnuje konkrétním aplikacím.

Abstract

This master thesis deals with the usage of OPC Data Access specification for access to physical device. Main task is to develop OPC server for I/O card Advantech PCI 1750. Server supports required parts of OPC Data Access 2.05. As for the optional parts, IOPCBrowseAddressSpace interface is implemented. A simple OPC client was also developed to demonstrate basic functionality of the server.

Another task of this thesis is to develop configuration interface (manager) for the server. Manager serves users in accessing some global options of the server. Those options are unavailable from OPC clients.

The theoretic part of this thesis contains a brief introduction to some types of interprocess communication in Windows. From the simpler ones to the COM model, which the OPC technology stands upon. Description of OPC Data Access specification is another theme of the theoretic part. The practical part pays attention to concrete applications.

České vysoké učení technické v Praze - Fakulta elektrotechnická

Katedra řídicí techniky ***Školní rok:*** 2004/2005

Z A D Á N Í D I P L O M O V É P R Á C E

Student: Ondřej Š ť a v í k
Obor: Technická kybernetika
Název tématu: OPC Server

Z á s a d y p r o v y p r a c o v á n í:

1. Prostudujte si OPC protokol a způsob tvorby klienta a serveru.
2. Navrhněte systémovou službu OPC server, který bude obsluhovat I/O kartu Advantech PCI.
3. Prostudujte otázku rozšíření OPC serveru o možnost zpracovávat periodické přerušení.
4. Server realizujte a vytvořte k němu i konfigurační prostředí pro něj v .NET C# a ukázkového klienta.

Seznam odborné literatury: Dodá vedoucí práce.

Vedoucí diplomové práce: Ing. Richard Šusta, Ph. D.
Termín zadání diplomové práce: zimní semestr 2004/2005
Termín odevzdání diplomové práce: leden 2006

L.S.

prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

prof. Ing. Vladimír Kučera, DrSc.
děkan

V Praze dne: 14.03.2005

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Využití řídících systémů v průmyslu se neustále rozrůstá. S nárůstem automatizace řídících procesů narůstá množství dat jak o řízeném procesu, tak o samotných zařízeních. Tyto data je potřeba předat klientům, tj. pracovním stanicím, které je budou využívat např. pro vizualizaci či řídící algoritmy. Technicky existuje více možností jak zařídit tento proces přenosu dat. Obvyklým způsobem bylo využití specifických ovladačů pro každé zařízení. To sebou přinášelo mnoho nevýhod. Mezi největší z nich patří

- 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.

Ve snaze co nejvíce eliminovat nejenom tyto nevýhody vznikly specifikace OPC (*Ole for Process Control*). O jejich údržbu a pravidelné aktualizace se stará organizace OPC Foundation, sdružující spoustu výrobců produktů pro řídící techniku. Tato diplomová práce se zabývá jednou z těchto specifikací, známou pod názvem OPC Data Access. Ta se využívá pro přístup k měřícím zařízením, akčním členům a dalším částem podílejících se na přímém řízení technologického procesu. Specifikací OPC Data Access se blíže zabývá kapitola 4.

OPC Data Access verze 2.05, popisovaná v této práci, funguje na operačním systému Windows. Využívá tedy metod meziprocesové komunikace navržené právě pro tento systém firmou Microsoft. Konkrétně se jedná o model COM (*Component Object Model*). Model COM, hojně využívaný nejen v řídící technice, ale obecně v každé složitější aplikaci pro systém Windows, je stručně popsán v kapitole 3.

OPC Data Access v zásadě popisuje komunikaci dvou stran - *serveru* a *klienta*. Server se stará o spolupráci s konkrétním zařízením a distribuci dat z tohoto zařízení klientům, kteří tyto data dále zpracovávají. Server může ale poskytovat uživatelům další výhody, které specifikace OPC Data Access nezahrnuje. Řešením může být vlastní rozšíření specifikace. Tato diplomová práce řeší tuto situaci jinak. Používá speciální rozhraní obsahující všechna dostupná nastavení serveru s vlastním komunikačním protokolem. Toto rozhraní, tzv. *manažer*, využívá jednu z jednodušších metod meziprocesové komunikace, konkrétně pojmenované roury. O pojmenovaných rourách a jim příbuzných metodách komunikace pojednává kapitola 2.

Kapitola 6 se věnuje konkrétním aplikacím této práce. OPC specifikace je v těchto aplikacích využita pro komunikaci s kartou Advantech PCI 1750. Tato měřící karta se hojně využívá v mnoha průmyslových aplikacích. Popis karty, instalace aplikací, jejich testy a ovládání je uvedeno v přílohách.

Kapitola 2

Základní typy meziprocesové komunikace ve Windows

Při komunikaci na bázi klient/server se musí obě zúčastněné strany domluvit jakým způsobem spolu budou komunikovat. Ne jinak tomu je při využití specifikace OPC v automatizaci. Komunikace COM, na které je OPC specifikace postavena, je jednou z metod meziprocesové komunikace v systému Windows. Existuje ale i řada dalších, jednodušších metod, které jsou ve Windows dostupné. Těmito metodami se zabývá tato kapitola. Od nejzákladnějších metod se postupně dostává až k předchůdcům modelu COM. Ten je pak detailněji popsán v kap. 3. Ostatní vyšší metody meziprocesové komunikace, jakými jsou hlavně v poslední době technologie navržené pro platformu .NET, popisuje lit. [2].

2.1 Namapované soubory

Technika namapovaných souborů (*Memory Mapped Files*) umožňuje procesu přístup do části paměti (na disku, v operační paměti...), jako kdyby byla součástí jeho adresního prostoru. Proces dostane ukazatel na tuto namapovanou oblast a pomocí něj může číst nebo měnit její obsah. Aby mohlo dojít k meziprocesní komunikaci, je tato oblast samozřejmě namapovatelná taky ostatními procesy. V případě přístupu více procesů současně je nutné použítí synchronizačních objektů, aby nedocházelo k porušení integrity dat. Namapované soubory jsou efektivní metodou lokální meziprocesové komunikace. Nelze je použít po síti. Více v lit. [13].

2.2 Poštovní schránky

Poštovní schránky jsou určeny pro pro posílání krátkých zpráv (dat) kratších než 424 bytů. Proces, který založí poštovní schránku je *poštovním serverem* a může číst zprávy které jsou mu doručeny *poštovním klientem*. Poštovní schránky se hodí hlavně k hromadnému rozesílání zpráv po jedné doméně v síti, ale mohou být použity i lokálně. Důvodem je fakt, že poštovní schránka je identifikována pomocí jména, pod tímto jménem však může existovat více jiných poštovních schránek na různých počítačích ve stejné doméně. Poštovní klient, který zná jenom jméno schránky, pak rozešle zprávu na všechny schránky pod tímto jménem. Poštovní server může být také zároven poštovním klientem, komunikaci lze tedy provozovat v obou směrech. Síťová komunikace je založena na aplikačním protokolu SMB (*Service Message Block*) viz lit. [13]. Nevýhodou poštovních schránek je, že protokol SMB v tomto případě interně využívá transportní protokol UDP (*User Datagram Protocol*), viz lit. [8]. Klient tedy nemá žádnou záruku, zda server skutečně obdržel zprávu. Jinak je tomu v případě pojmenovaných rour, viz podkap. 2.4. Více v lit. [13].

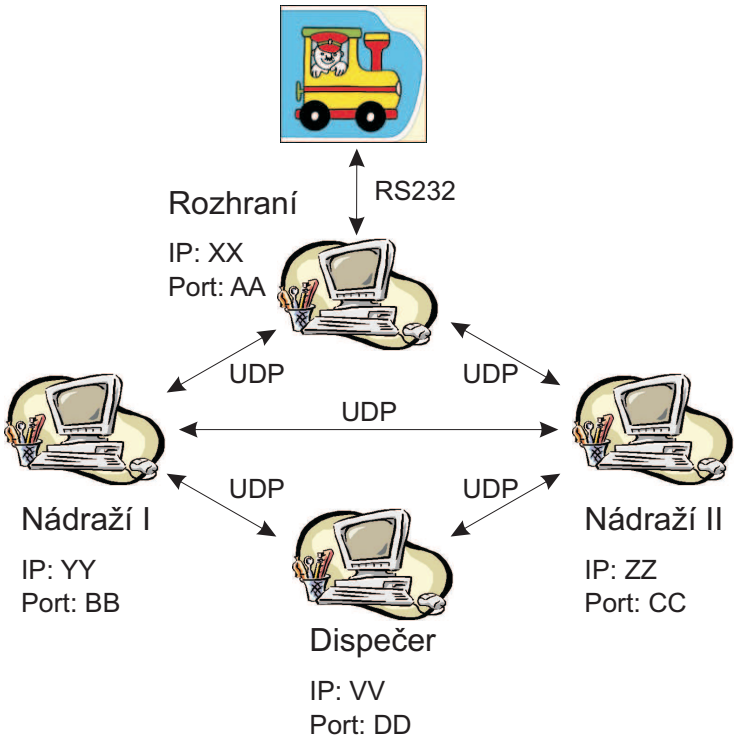
2.3 Sockety

Komunikace pomocí socketů je také založena na posílání zpráv mezi klientem a serverem. Délka zprávy není při této komunikaci omezena, zprávy jsou ale transportním protokolem rozděleny na menší fragmenty, tzv. *packety*. Ty mívají max. velikost okolo 64kB. Je primárně určena pro komunikaci po nejrozsáhlejších sítích WAN (*World Area Network*), lze ji ale použít i na lokálním počítači. Obrovskou výhodou je nezávislost na síťovém transportním protokolu. Může běžet na TCP (*Transmission Control Protocol*), UDP, IPX (*Internetwork Packet EXchange*) apod. Klient a server také mohou běžet v odlišných operačních systémech, jako je např. Linux.

Socket je komunikační kanál, podobný rouře, viz podkap. 2.4. Může být založený na lokálním počítači, nebo mezi dvěma počítači v síti. Každý konec socketu je jednoznačně identifikován IP adresou, transportním protokolem a portem, viz lit. [9]. Klient a server pak pracují ze socketem podobně jako z obyčejným souborem.

Posílání krátkých zpráv pomocí socketů může být využito při řízení jednoduchých procesů a je použita např. při řízení modelu vlaku na katedře řízení FEL ČVUT, viz obr. 2.1.

Příklad 2.1 (Řízení modelu vlaku pomocí UDP socketů):



Obrázek 2.1: Použití UDP socketů

Dispečer, obě nádraží a rozhraní mají dohodnutou strukturu zpráv posílaných UDP sockety. Dispečer určuje jízdní řád. Nádraží si ho od něj přejímají a koordinují navzájem polohu jedoucích a odstavených vlaků. Výsledné nastavení semaforů a výhybek posílají do rozhraní, které ovládá model vlaku.

2.4 Roury

Pod pojmem roura si lze představit část sdílené operační paměti, které se dá použít k výměně dat mezi procesy. Tento kus paměti je tzv. *pseudo* souborem. Operační systém se k němu chová tak jako by se jednalo o normální soubor, pouze některé I/O funkce pro práci se soubory mají v případě práce s rourou lehce odlišné chování.

Na jednom konci roury do ní zapisuje jeden proces a na druhém z ní čte jiný proces. Data jsou čtena ve stejném pořadí jako byla zapsána, roura má tedy FIFO (*First In, First Out*) architekturu.

Existují dva základní druhy rour

- anonymní (*Anonymous*) - pouze pro komunikaci v jednom směru. Nejsou identifikovány pomocí jména, ale pomocí *handles* (čísel). Pokud má komunikace běžet v obou směrech, je potřeba vytvořit roury dvě. Nelze je používat při komunikaci na vzdálených počítačích.
- pojmenované (*Named*) - identifikovány podle jména. Možnost použití na vzdálených počítačích a v obousměrném režimu. Pro přístup k vzdáleným počítačům je použit aplikační protokol SMB, který interně využívá transportní protokol TCP/IP.

Pro spojení manažera a OPC serveru jsem si vybral pojmenované roury, z důvodů uvedených v podkap. 6.2.

2.4.1 Pojmenované roury

Pojmenované roury fungují podobně jako telefonování. Rouru založí jeden proces (server), z druhé strany se k ní připojí jiný proces (klient). Tyto dva procesy pak mezi sebou přenáší data přes komunikační kanál. Informace o poloze začátku a konce roury se nastaví při jejím vytvoření a nemusí být posílány s každým kusem přenášených dat. Klient a server při komunikaci používají standardní I/O funkce Win API pro práci se soubory a dále funkce dedikované jen pro roury.

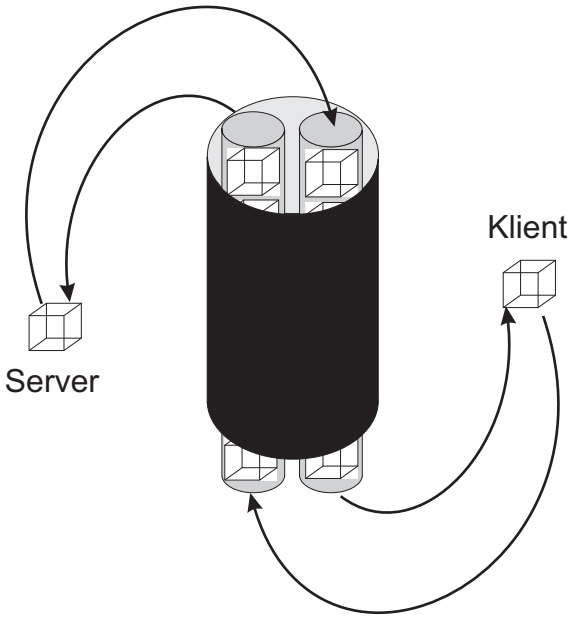
2.4.1.1 Jméno roury

Jméno pojmenované roury má předepsaný formát

```
\\servername\pipe\pipename
```

Klient vyplní pole *Servername* jménem vzdáleného počítače. Je-li server na stejném počítači, vloží „.“. Server při vytváření roury vloží vždy „.“, jelikož nelze vytvářet roury na vzdálených počítačích.

Jméno roury (*pipename*) může obsahovat jakékoliv znaky kromě zpětného lomítka a může být dlouhé max. 256 znaků.



Obrázek 2.2: Ilustrační schéma roury

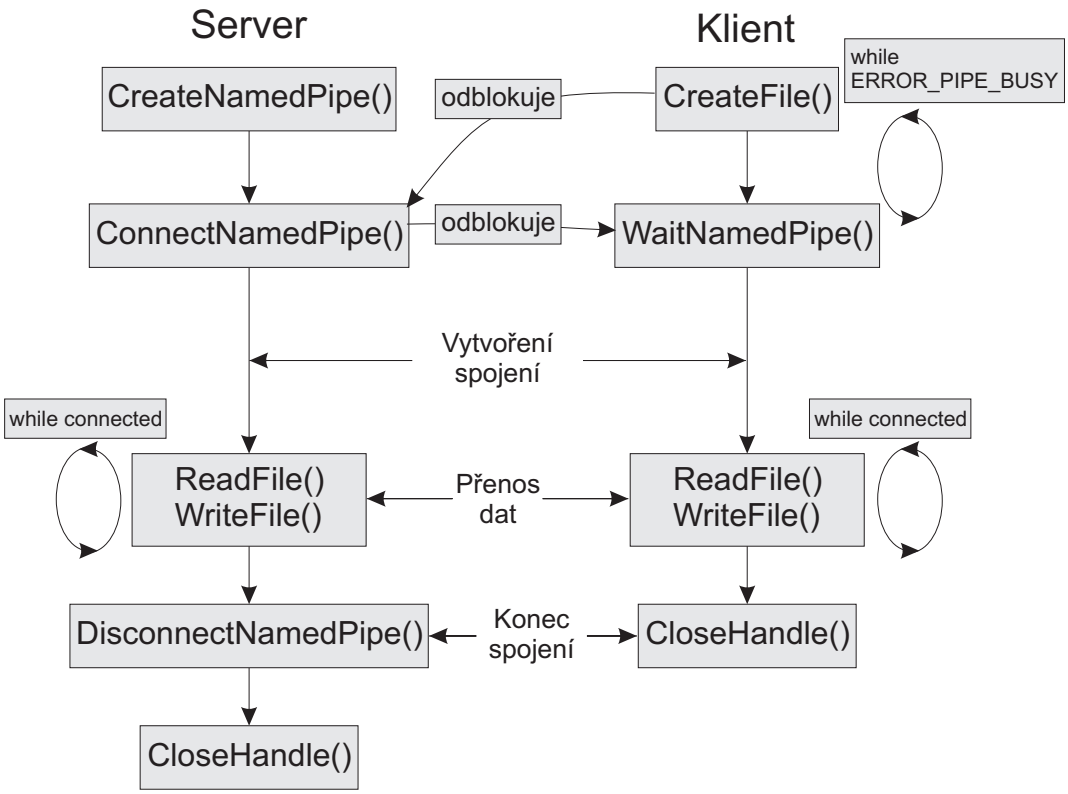
2.4.1.2 Základní funkce

Základní funkce Win API pro roury jsou uvedeny v tabulce 2.1.

Funkce	Popis
CreateNamedPipe	Vytvoří instanci roury na straně serveru.
ConnectNamedPipe	Pomocí této funkce server čeká na příchozí spojení.
CreateFile	Pokusí se navázat spojení se serverem.
WaitNamedPipe	Klient volá tuto funkci v čekací smyčce na spojení. Jestliže obdrží pozitivní odezvu od serveru, opustí čekací smyčku.
ReadFile	Přečte data z roury.
WriteFile	Zapíše data do roury.
DisconnectNamedPipe	Odpojí rouru na straně serveru.
CloseHandle	Zruší instanci roury.

Tabulka 2.1: Základní funkce pro práci s rourami

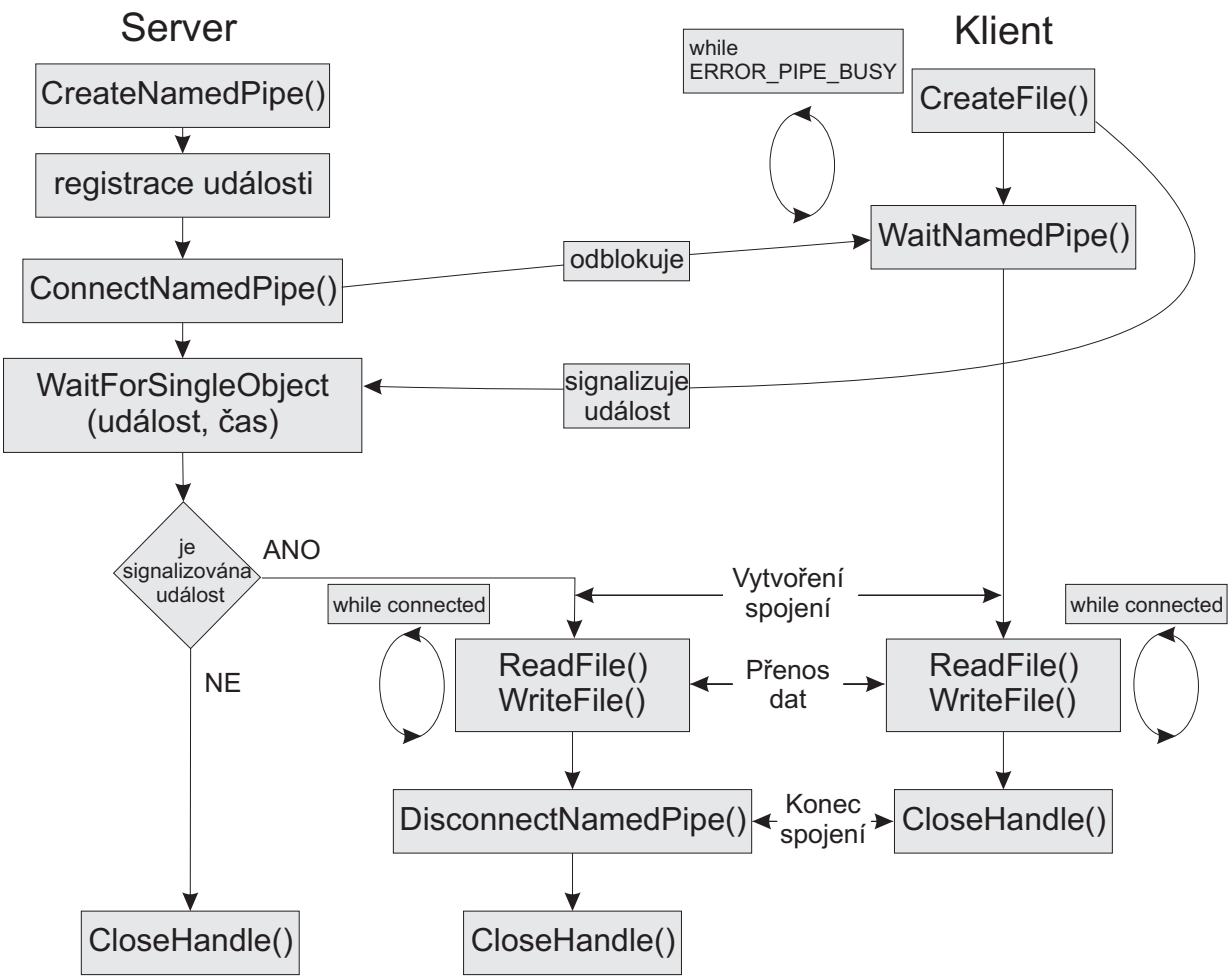
Na obr. 2.3 je postup klienta a serveru při komunikaci pojmenovanou rourou.



Obrázek 2.3: Komunikace serveru a klienta pomocí pojmenované roury

2.4.1.3 Neblokovací režim

Vlákna na straně serveru/klienta se mohou dostat do nepříjemné situace. Pokud na klientské straně nedojde k volání CreateFile, funkce ConnectNamedPipe blokuje vlákno serveru na nekonečně dlouhou dobu. Podobná situace nastane u funkce Read, když v rouře nejsou žádná data, nebo u funkce Write, když je roura plná. Tato situace se dá obejít použitím speciální datové struktury *OVERLAPPED*. Pomocí této struktury si v systému zaregistrujeme událost (*event*). Tato událost bude signalizovat, zda došlo na klientské straně k připojení. Výhodou použití události je, že se na ni dá čekat konečně dlouhou dobu a pak pokračovat v dalším kódu programu. K čekání na události a podobné synchronizační objekty se v systému Windows používá funkce *WaitForSingleObject*. Řízení iniciace připojení pomocí události je ilustrováno na obr. 2.4. Více v lit. [13].



Obrázek 2.4: Komunikace serveru a klienta pomocí pojmenované roury s použitím události

2.5 RPC

RPC (*Remote Procedure Call*) umožňuje jednomu procesu (klientovi) vzdáleně volat funkce jiného procesu (serveru) na lokálním či vzdáleném počítači. Tato technologie se využívá také v modelu DCOM, což je nadstavba modelu COM, viz kap. 3. RPC server a klient opět mohou pracovat na jiných platformách (UNIX, MacOS…) a využívat různé transportní protokoly. Více v lit. [13].

2.6 Schránka

Schránka (*Clipboard*) je sdílená paměť mezi všemi oknovými aplikacemi Windows. Přístup ke schránce umožňují aplikacím Windows zprávy. Aplikace nemůže použít tento typ komunikace bez toho, aniž by o tom uživatel aplikace věděl. Vývojář aplikace, která podporuje schránku, implementuje pouze funkce potřebné pro přenos povelů uživatele (např. Copy, Paste, apod.) do formy Windows zpráv. Jelikož různé aplikace používají různé formáty dat, obsah schránky pro ně nemusí být vždy čitelný. Uživatel řídící komunikaci by si měl být vědom toho, zda aplikace, která čte data ze schránky podporuje formát dat aplikace, která data do schránky uložila. Schránku lze použít jen lokálně. Více v lit. [13].

2.7 DDE

DDE (*Dynamic Data Exchange*) je vylepšením schránky. Je tedy založen čistě na posílání Windows zpráv.

Aplikace, která inicializuje spojení, je DDE klientem. Aplikace, která odpovídá, je DDE serverem. Klient a server jsou účastníci DDE *konverzace*. Klient, resp. server mohou vézt více než jednu konverzaci s různými servery resp. klienty. Pro každou z nich ale musí vytvořit jedno okno. Mezi konverzacemi lze podle potřeby přepínat. Komunikace mezi oběma stranami může probíhat v obou směrech.

Každou konverzaci charakterizují tři objekty

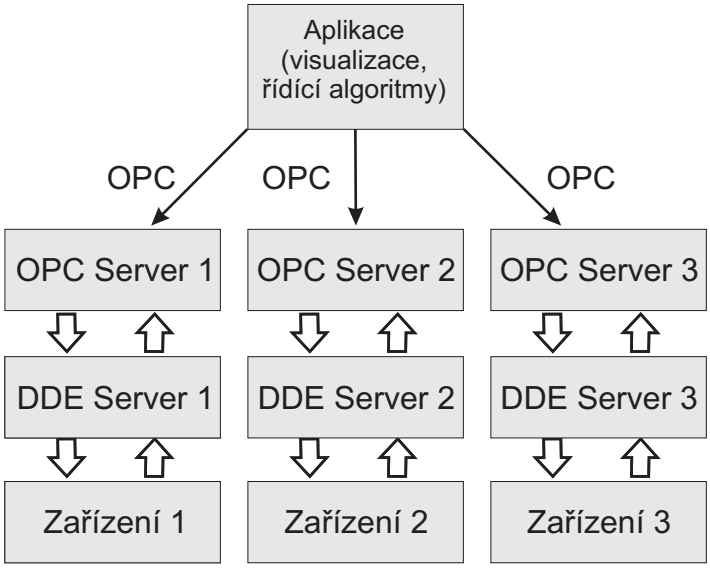
- jméno aplikace (*application*) - identifikuje server, např. „Excel“.
- téma konverzace (*topic*) - určuje objekt v rámci kterého se budou data přenášet, např. pro aplikace operující s dokumenty je to obvykle jméno souboru. Spolu se jménem aplikace jednoznačně indentifikuje konverzaci.
- položka (*item*) - konkrétní data vztahující se ke konverzaci. Lze použít jakýkoliv z předdefinovaných formátů dat používaných pro schránku, nebo nadefinovat vlastní formát dat.

Klient také může realizovat jedno nebo více *permanentních* spojení. Permanentní spojení slouží pro automatické obeznámení klienta o změně datové položky. Toto spojení může být realizováno dvěma způsoby

- server pošle pouze oznámení o změně dat ale konkrétní data posílá až na vyžádání klientem.
- server pří změně dat posílá automaticky vše.

Komunikaci pomocí DDE lze použít i na vzdálených počítačích pomocí služby NetDDE (NETDDE.EXE), která musí běžet na obou počítačích účastnících se konverzace. DDE po síti využívá transportní protokol TCP/IP.

DDE protokol našel také široké uplatnění v řídící technice, kde se používá hlavně k přístupu k hardware. DDE server pak pomocí ovladače konkrétního hardware zprostředkovává data klientům. V dnešní době se již DDE protokol v řízení samostatně příliš nevyužívá. Byl překonán standardem OPC založeném na komponentovém modelu COM, o kterém pojednávají další kapitoly. Bývá ale využíván v řadě OPC serverů jako mezičlánek komunikace, viz obr. 2.5. Více o DDE v lit. [13] a [14].



Obrázek 2.5: Využití DDE v OPC

2.8 OLE

Technologie OLE (*Object Linking and Embedding*) byla uvedena ve dvou dost odlišných verzích.

Verze OLE 1.0 je založena na protokolu DDE, tedy další posílání Windows zpráv. Byl navržen pro vkládání objektů (dokumentů, obrázků...) do objektů jiných aplikací (např. tabulka Excelu do dokumentu Word). Při pokusu o editaci vloženého objektu je automaticky spuštěna instance původního programu (Excel), pokud již neběží. Objekt pak šlo editovat ve své mateřské aplikaci. To by bylo teoreticky možné i pomocí samotného DDE. Ten ale vyžaduje, aby instance původní aplikace byla již aktivní. Využití DDE se ale následně ukázalo jako velice nevhodné a pomalé. Proto byla technologie OLE přepracována.

Filozofie OLE 2.0 obsahovala již myšlenku komponentového software. Klient (Word) již mohl využívat služby serveru (Excel) ve svém vlastním prostředí. Tuto funkcioanalitu umožňovaly už předtím také klasické DLL (*Dynamic Linked Library*) knihovny, které ukrývaly funkce serveru. U nich se ale naráželo na mnoho problémů, viz lit. [7]. Technologie OLE 2.0 všechny tyto problémy řešila. Byla ale dost komplikovaná a vhodná pouze pro určitý typ úloh. Proto byly základní principy zjednodušeny, zobecněny a vznikl model COM, viz. kap. 3.

Kapitola 3

Technologie COM

3.1 Úvod

Komponentový model COM (*Component Object Model*) je jednou z možných metod meziprocesové komunikace ve Windows. Použití COM má také další výhodu, kterou je rozdělení původně pouze monolitické aplikace, tj. aplikace tvořené jedním spustitelným souborem obsahující veškerý kód, na jednoznačně oddělitelné části. Jistým způsobem jak vyřešit modularitu programu jsou dynamicky linkované knihovny DLL, se kterými přišla firma Microsoft poprvé v 16bitových Windows. Z důvodů různých nekompatibilit, viz lit. [7], toto řešení přestalo vyhovovat.

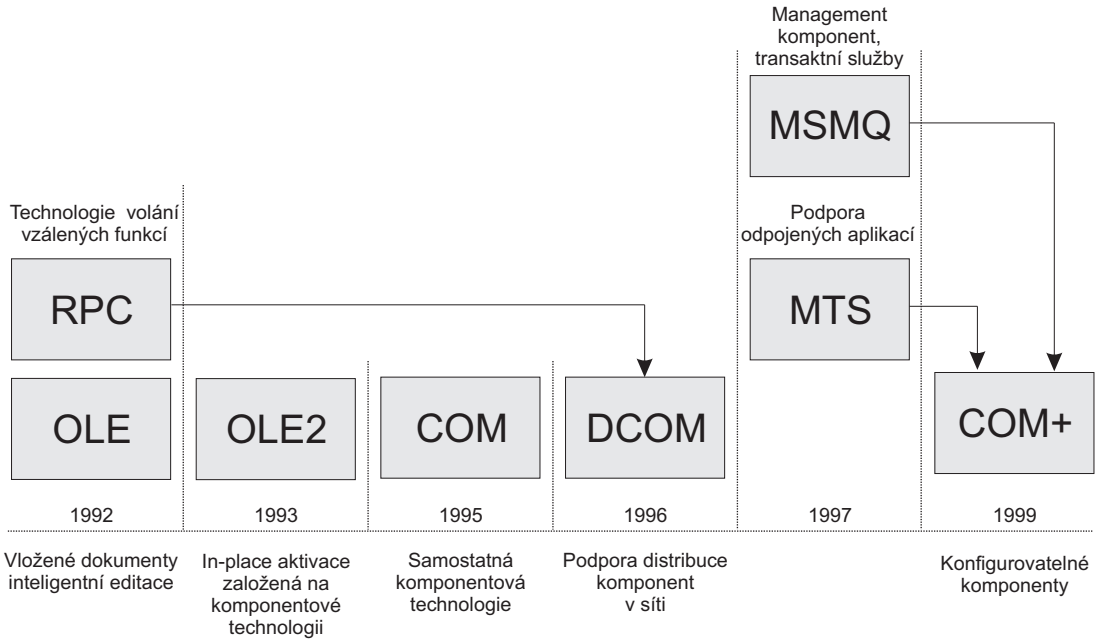
Nebude-li doplněno jinak, detailnější informace o technikách popsaných v této kapitole lze přehledně najít v lit. [1].

Komponentový objektový model COM se poprvé objevil v roce 1995. Jeho předchůdci byly modely OLE a OLE2, viz podkap. 2.8. COM se stal základem vývoje programového vybavení pro platformu Windows. Rozvoj komponentové tvorby softwaru ale tímto krokem neustal. COM technologie se stala také základem pro mnoho dalších jako jsou např.

- DCOM (*Distributed COM*), poskytuje nadstavbové služby pro pro zavádění a komunikaci s komponentami na vzdálených počítačích.

- COM+, postavena na COM a DCOM s využitím dalších služeb : MTS (*Microsoft Transaction Server*), prostředí řešící problémy současné obsluhy více klientů, řízení bezpečnosti, administrace a robustního ošetření chybových stavů a MSMQ (*Microsoft Message Queue Server*), produkt který umožňuje zaznamenávat požadavky na straně klienta do tzv. front zpráv a spolehlivě je přenášet na stranu serveru i při poruchách sítě.

- OPC (*OLE for Process Control*) průmyslový standard postavený na COM, používaný pro zprostředkování a archivaci dat v řízení procesů. Tomuto standardu a jeho implementaci i konkrétní aplikaci se věnuje tato práce v pozdějších kapitolách.

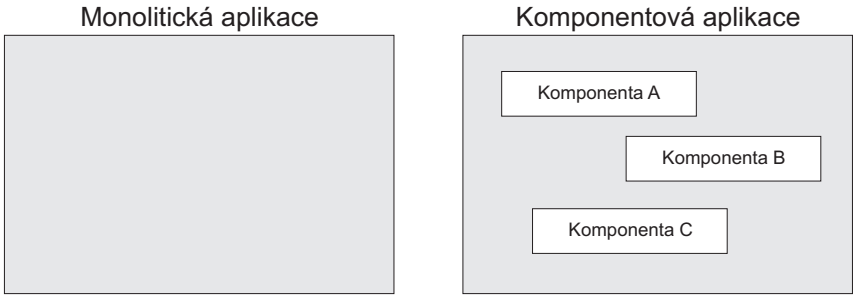


Obrázek 3.1: Historie komponentových modelů

3.2 Komponenty, rozhraní

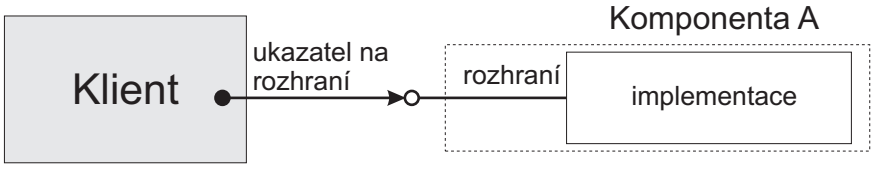
Pod pojmem komponenta se rozumí jednoznačně oddělitelná část aplikace. Ve srovnání s monolitickými aplikacemi lze komponentovou aplikaci schématicky znázornit třeba jako na obr. 3.2.

Každá komponentová aplikace, neboli *COM server*, poskytuje pomocí instancí svých komponent služby jedné nebo více jiným aplikacím, tzv. *COM klientům*. To, co musí splňovat klient a server, i zajištění mechanismu jejich vzájemného propojení, popisuje právě model COM. Každou komponentu COM lze logicky rozdělit na dvě části, *Rozhraní (Interface)* a *Implementaci*. Komponenta může obsahovat více rozhraní a k nim odpovídajících implementací. Z pohledu programovacích jazyků je interface seznamem metod a implementace definicí výkonných těl těchto metod. Rozhraní nesmí obsahovat ani žádná



Obrázek 3.2: Monolitická a komponentová aplikace

členská data, která by mohla způsobit nekompatibilitu komponentové aplikace (jako se tomu často stává u klasických DLL knihoven, kde je vše dohromady). Klientská aplikace pak získává pouze ukazatel na rozhraní dané instance komponenty uvnitř serveru, aniž by něco věděla o implementaci tohoto rozhraní.



Obrázek 3.3: Spojení klienta s instancí komponenty

Příklad rozhraní a implementace je uveden níže. Příklad je napsaný v jazyce C++, jakož i většina budoucích příkladů, jelikož je tento jazyk použit při tvorbě konkrétních COM komponent v této práci. V jazyce C++ lze nejlépe popsat rozhraní pomocí *abstraktní* třídy

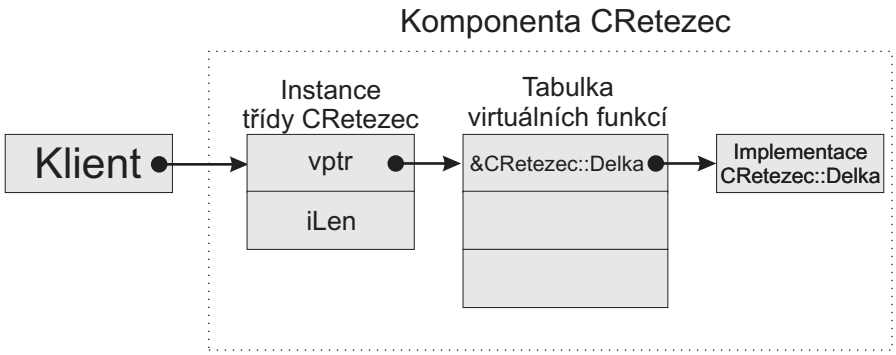
Příklad 3.1 (Deklarace a implementace rozhraní komponenty):

```
class IRetezec
{
    public:
        virtual int Delka() = 0;
}
```

Implementace rozhraní může vypadat takto :

```
class CRetezec : public IRetezec
{
      int iLen;
    public:
        int Delka() {return iLen;};

//konstruktor, destruktor
...
}
```

Interně je tato komunikace založena na *tabulkách virtuálních funkcí*. Tabulku virtuálních funkcí má každá abstraktní třída. Pro každou třídu existuje pouze jedna a je sdílena všemi instancemi třídy. Každá instance pak vlastní na tabulku ukazatel. Samotná tabulka obsahuje ukazatele na implementace všech virtuálních metod dané třídy. Klient pak ve skutečnosti získává ukazatel na ukazatel na tabulku virtuálních funkcí. Situace je na obr. 3.4.



Obrázek 3.4: Realizace spojení klienta a instance komponenty

V každém programovacím jazyce umožňujícím tvořit komponenty se deklarace rozhraní provádí různě a ostatní jazyky této deklaraci nerozumí. Proto je nutno vytvořit tzv. *binárně kompatibilní* rozhraní. Binární kompatibilita znamená, že komponenty mohou být po svém přeložení využívány i klienty vytvořenými v jiných programovacích jazycích. O tom, jak generovat binárně kompatibilní rozhraní pojednává podkap. 3.4.

3.3 Rozhraní COM

Příklad 3.1 ukazuje jak může vypadat deklarace rozhraní komponenty. Aby tato deklarace vyhovovala pravidlům COM, musí být ještě upravena.

Příklad 3.2 (Deklarace rozhraní COM):

```
class IRetezec : public IUnknown
{
    public:
        virtual HRESULT __stdcall  Delka( int *piVysledek) = 0;
}
```

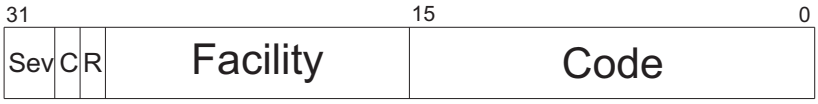
Základními rozdíly jsou

- volací konvence *__stdcall*.
- návratová hodnota funkce typu *HRESULT*.
- podmínka implementace rozhraní *IUnknown*.

Volací konvence __stdcall říká, jak budou funkci předávány parametry a dále pak, kdo bude parametry odstraňovat(volající, nebo volaná strana). V jazyce C++ je nutno ji použít, pokud chceme zajistit kompatibilitu s ostatními programovacími jazyky.

3.3.1 Návratové kódy

Návratová hodnota funkce typu HRESULT je 32bitové číslo obsahující kód, který popisuje úspěšnost metody. Všechny kódy používané v systému Microsoft Windows lze najít v hlavičkovém souboru *winerror.h*. Pokud nevyhovují, nebo je potřeba některé dodefinovat, lze vytvořit vlastní hlavičkové soubory, např. v OPC je to *OPCError.h*. Struktura čísla HRESULT je na obr. 3.5



Obrázek 3.5: Struktura kódu HRESULT

Bity C (*Customer Code Flag*), resp. R (*Reserved Bit*) jsou rezervovány pro Microsoft a mají konstantní hodnotu 1 , resp. 0. Bity Sev (*Severity Code*) můžou nabývat 4 hodnot : Success(0), Informational(1), Warning(2), Error(3). Bity zdroje události (*Facility*) musí být pro uživatelské kódy nastaveny na hodnotu FACILITY_ITF(4), zbytek je rezervován pro Microsoft. Z bitů pro vyjádření konkrétní chyby (*Code*), lze pro vlastní návratové hodnoty použít rozsah 0x0200-0xFFFF. Každý návratový kód by měl mít nadefinovanou symbolickou hodnotu, která obvykle obsahuje první písmeno nebo dokonce celé slovo z hodnot Sev, např. S_OK (0x00000000), E_NOINTERFACE (0x80004002). Tyto návratové hodnoty používá většina metod, ale jsou i výjimky, např. metody AddRef a Release o kterých se píše v podkap. 3.3.2. Původní návratovou hodnotu funkce typu int (*integer*) z příkladu 3.1 lze nyní předat odkazem přes parametr *piVysledek*.

3.3.2 IUnknown

Rozhraní IUnknown je základním rozhraním každé komponenty. Řídí životnost instance komponenty a umožňuje klientům dotázat se na rozhraní, která komponenta podporuje. Každá komponenta musí implementovat metody IUnknown uvedené v tab. 3.1.

Metoda	Popis
AddRef	Inkrementuje počítadlo referencí, neboli počet odkazů na instanci komponenty od všech připojených klientů. Metoda vrací aktuální počet referencí.
Release	Dekrementuje počítadlo referencí. Pokud počítadlo klesne na hodnotu 0, komponenta se může sama odstranit z paměti. Metoda vrací aktuální počet referencí.
QueryInterface	Slouží k dotazu na existující rozhraní implementované komponentou.

Tabulka 3.1: Seznam metod rozhraní IUnknown

Nyní lze rozvést příklad implementace rozhraní z př. 3.2.

Příklad 3.3 (Implementace rozhraní COM):

```
class CRetezec : public IRetezec
{
      int   iLen;
      ULONG ulRefCount;
    public:
//metody rozhrani  IRetezec
        HRESULT __stdcall Delka( int *piVysledek);

//metody rozhrani IUnknown
        ULONG   __stdcall AddRef( VOID);
        ULONG   __stdcall Release( VOID);
        HRESULT __stdcall QueryInterface ( REFIID riid, LPVOID *ppvObject);
//konstruktor, destruktor
...
}
```

Implementace konstruktoru a destruktoru je závislá na realizaci komponenty a na jejím apartmentu, viz podkap. 3.5. Ostatní metody lze implementovat třeba takto :

```
HRESULT __stdcall CRetezec::Delka ( int *piVysledek)
{
    if (!piVysledek)
        return E_INVALIDARG;
    *piVysledek = iLen;
    return S_OK;
}

ULONG   __stdcall CRetezec::AddRef ( VOID)
{
    ++ulRefCount;
    return ulRefCount;
}
```

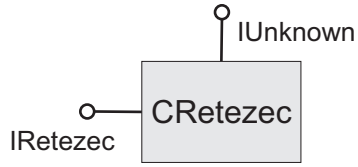
```
ULONG   __stdcall CRetezec::Release( VOID)
{
    if (--ulRefCount != 0)
        return ulRefCount;
    delete this;
    return 0;
}

HRESULT __stdcall CRetezec::QueryInterface( IID riid, LPVOID *ppvObject)
{
    if (!ppvObject)
        return E_INVALIDARG;

    if (riid == IID_IRetezec)
    {
        *ppvObject = (IRetezec*) this;
    }
    else if (riid == IID_IUnknown)
    {
        *ppvObject = (IUnknown*) this;
    }
    else
    {
        *ppvObject = NULL;
        return E_NOINTERFACE;
    }
    AddRef();
    return S_OK;
}
```

Metoda QueryInterface (nezávsile na apartmentu a realizaci) musí splňovat následující požadavky

- rozhraní IUnknown je jedinečné, tzn. vždy získáme stejné rozhraní IUnknown.
- pokud se nám podaří pomocí QueryInterface získat ukazatel na rozhraní, musí se to

podařit znovu i v budoucnu. Naopak, pokud jednou metoda při dotazu na rozhraní skončí neúspěšně, ani v budoucnu úspěšná nebude.

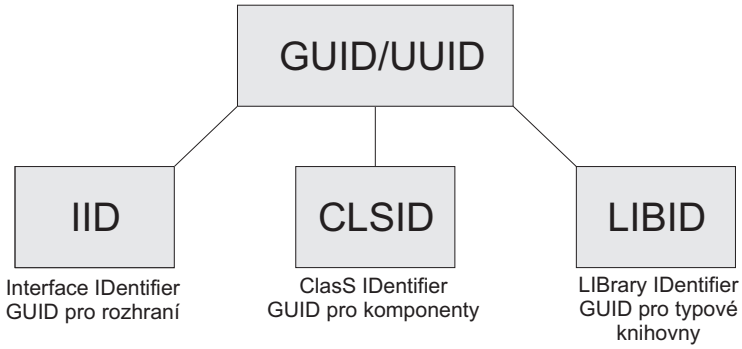
- metoda je reflexivní, tzn. že si můžeme vyžádat ukazatel na rozhraní, které již máme.
- metoda je symetrická, tzn. že se vždy můžeme vrátit k rozhraní ze kterého jsme vyšli. Máme-li např. ukazatel na rozhraní IUnknown a pomocí jeho metody QueryInterface získáme rozhraní IRetezec, musíme být schopni pomocí rozhraní IRetezec a jeho metody QueryInterface získat zpátky ukazatel na rozhraní IUnknown.
- metoda je tranzitivní, tzn. že se vždy můžeme dostat k danému rozhraní z jiného libovolného rozhraní téhož objektu, pokud jsme se k němu již odněkud dostali.
- metoda automaticky inkrementuje počitadlo referencí voláním metody AddRef. O dekrementaci počitadla se pak musí postarat klient voláním Release.



Obrázek 3.6: Schematická značka komponenty CRetezec

3.3.3 GUID

Příklad 3.3 používá v metodě QueryInterface parametr *riid*, který je typu IID. IID (*Interface IDentifier*) je datová struktura zastupující celosvětově jedinečné 128bitové číslo. Obecně se takovému číslu říká GUID (*Globally Unique IDentifier*) či UUID (*Universally Unique IDentifier*). Zkratka IID se většinou používá v kontextu s identifikací rozhraní. Nejen rozhraní, ale i komponenty a další objekty musí mít své GUID, pro každý z nich se ale číslu GUID říká jinak. Záleží jen na typu objektu, jaká zkratka se použije (viz obr. 3.7), struktura čísla je ale stejná. Firma Microsoft nabízí různé nástroje, které umí generovat GUID. Jsou to např. programy *guidgen.exe* nebo *uuidgen.exe*.



Obrázek 3.7: Přehled identifikačních čísel

3.4 Jazyk IDL

Jazyk IDL (*Interface Definition Language*) byl původně navržen organizací OSF (*Open Software Foundation*), která ho využívala pro metody vzdáleného spouštění funkcí RPC, viz podkap. 2.5. IDL i RPC byly firmou Microsoft přejaty, rozšířeny a využity v technologiích COM a DCOM. IDL není programovacím jazykem v pravém smyslu slova. Slouží pouze k popisu rozhraní komponenty a k přeložení tohoto rozhraní do binárně kompatibilní formy. Vlastní implementace rozhraní komponenty se již provede v C++, Java, VB apod. Následující příklad ukazuje jak lze definici rozhraní z př. 3.2 přepsat v jazyce IDL.

Příklad 3.4 (Deklarace rozhraní COM v jazyce IDL):

```
import "unknnw.idl";
[object,uuid (123445678 - 1234 - 0000 - abcd - 0000000001)]
interface IRetezec:IUnknown
{
    HRESULT Delka ([out, retval] int *piVysledek);
}
```

Definice se skládá z těchto hlavních částí

- výraz *import "unknnw.idl";* vloží do souboru obsah souboru *unknnw.idl*, což je IDL popis rozhraní IUnknown.
- atributy rozhraní uvedené v hranatých závorkách : *object* - říká že definujeme rozhraní, *uuid* - IID identifikující rozhraní, viz podkap. 3.3.3.

- klíčové slovo *interface*, název rozhraní, dvojtečka a jméno rodičovského rozhraní.
- složené závorky obsahující deklarace jednotlivých metod rozhraní, doplněných atributy. Základní z nich jsou uvedeny v tab. 3.2.

Atribut	Význam
[in]	Data jsou naplněna volající stranou a jsou přenesena pouze v jednom směru - od klienta ke komponentě
[out]	Data jsou naplněna volanou stranou a přenáší se od komponenty ke klientovi
[in, out]	Parametr je přenášen oběma směry
[out, retval]	Stejné jako out, jen s bližším označením návratové hodnoty pro jazyky jako VB, Java

Tabulka 3.2: Základní typy parametrů pro deklarace metod v jazyce IDL

Takový obsah IDL souboru ale ještě nestačí k vygenerování binárně kompatibilního rozhraní, tzv. *typové knihovny*. Do souboru je ještě nutno přidat následující řádky

Příklad 3.5 (Deklarace typové knihovny v jazyce IDL):

```
[uuid (12345678-0000-0000-0000-0000000003),
 helpstring("Retezec Type Library"),
 version (1.0)]

library Retezec
{
    importlib("stdole32.tlb");
    interface IRetezec;



    [uuid(12345678-0000-0000-0000-000000002]
    coclass CGenerator
    {
        interface IGenerator;
    }
}
```

Hlavní části jsou tyto

- skupina atributů knihovny v hranatých závorkách: *uuid* - LIBID, viz podkap. 3.3.3, identifikující typovou knihovnu, *helpstring* - výraz blíže definující jméno typové knihovny. Tento výraz se pak zobrazuje v různých OLE prohlížečích, *version* - výraz definující verzi knihovny, taktéž zobrazovaný v OLE prohlížečích.
- klíčové slovo *library* a název souboru knihovny - typová knihovna pak bude uložena v souboru *"Retezec.tlb"*, vše pod tímto řádkem bude součástí knihovny.
- klíčové slovo *importlib* s názvem již hotové knihovny - vloží knihovnu *stdole32.tlb*, která obsahuje definice standardních rozhraní (IUnknown apod.).
- klíčové slovo *interface* s názvem rozhraní - vloží do knihovny definici rozhraní.
- atributy komponenty v hranatých závorkách: *uuid* - CLSID identifikující komponentu.
- klíčové slovo *coclass* definující komponentu CGenerator, ve složených závorkách je pak seznam rozhraní, která komponenta implementuje.

Toto samozřejmě nejsou jediné příkazy, které může IDL soubor implementovat. Další klíčová slova lze najít v lit. [10]. K vygenerování typové knihovny již stačí jen zkompilovat IDL soubor kompilátorem MIDL. Kompilátor vygeneruje tyto soubory

- Retezec.h - hlavičkový soubor pro C/C++.
- Retezec_i.c - definice GUID konstant, viz podkap. 3.3.3.
- Retezec_p.c - kód pro objekty proxy/stub, viz podkap. 3.5.2.
- dlldata.c - data marshalling kód.
- Retezec.tlb - typová knihovna.

První dva soubory se obvykle používají pří implementaci rozhraní komponenty v C/C++. Všechny čtyři soubory se taktéž dají použít k vytvoření *standardní marshalovací DLL knihovny*, viz podkap. 3.5.2.

Typová knihovna tak vlastně určuje, jak bude vypadat virtuální tabulka funkcí pro dané rozhraní. Pro rozhraní IRetezec by vypadala jako na obr. 3.4. Existují programovací jazyky, které nedokáží před zpracováním kódu typovou knihovnu načíst. Jedná se hlavně

o skriptovací jazyky (VBScript, JScript apod.), dále VBA (*Visual Basic for Applications*), ale i rané verze Visual Basicu. Aby i tyto jazyky mohly využít metod komponenty, musí komponenta implementovat rozhraní *automation*. Automation rozhraní je dobře popsáno v lit. [1].

3.5 Realizace COM serveru

COM server může být realizován jako takzvaný in-process (DLL, OCX, AX,... soubor) nebo out-of-process (EXE soubor). In-process server ke svému životu potřebuje vždy nějaký hostitelský proces. Běží-li in-process server na stejném počítači jako klientská aplikace, je tímto procesem klientská aplikace, běží-li na vzdáleném počítači, vytvoří se speciální proces zvaný *surrogate*. Out-of-process server má vždy svůj vlastní proces.

3.5.1 Apartmenty

Každá komponenta potřebuje ke svému životu *Apartment*, tedy jakýsi „byteček“. Pro Apartmenty platí

- apartment je skupina objektů(instancí komponent) v rámci procesu.
- každý objekt náleží právě do jednoho procesu.
- každý proces může obsahovat několik apartmentů.
- apartmenty se používají pro mapování vláken k objektům.

Objektový model COM definuje dva základní typy apartmentů

- STA (*Single-Threaded Apartment*) - může obsahovat pouze jedno výkonné vlákno. Kód jednotlivých komponent umístěných v tomto apartmentu není vykonáván pararelně a volání metod komponent jsou serializovány do formy Windows zpráv. Tak je automaticky zajištěno jejich postupné vykonávání a synchronizace.
- MTA (*Multi-Threaded Apartment*) - metody komponent v tomto apartmentu můžou být vykonávány více vlákny zároveň. Na komponenty jsou v tomto případě kladeny větší požadavky a musí být více-vláknově bezpečné.

Můj COM server je realizován jako out-of-process a využívá MTA apartment. Důvody jsou uvedeny v podkap. 6.1. Každý proces může obsahovat pouze jeden MTA apartment. První vlákno procesu, které chce vstoupit do MTA apartmentu, zároveň tento apartment založí. Pro práci s apartmentem existují funkce Win API uvedené v tab. 3.3.

Atribut	Význam
CoInitializeEx	Přiřadí vláknu apartment. V dalším kódu pak musí být někdy volaná párová inverzní operace CoUninitialize.
CoUninitialize	Odebere vláknu apartment.

Tabulka 3.3: Funkce pro založení apartmentu

Předávání ukazatelů a parametrů mezi klientskou aplikací a out-of-process serverem není jednoduchá záležitost, jelikož mají oba procesy různý adresový prostor. Dochází zde k přeměně parametrů do nezávislé přenositelné podoby , tzv. *marshalování parametrů*. Obecně k je potřeba zajistit marshalování parametrů i v rámci jednoho procesu (v případě in-process komponent) vždy když je klientské vlákno umístěno v apartmentu, který není kompatibilní s apartmentem, v němž byla vytvořena instance komponenty. Kompatibilita apartmentů je pěkně popsána v lit. [1].

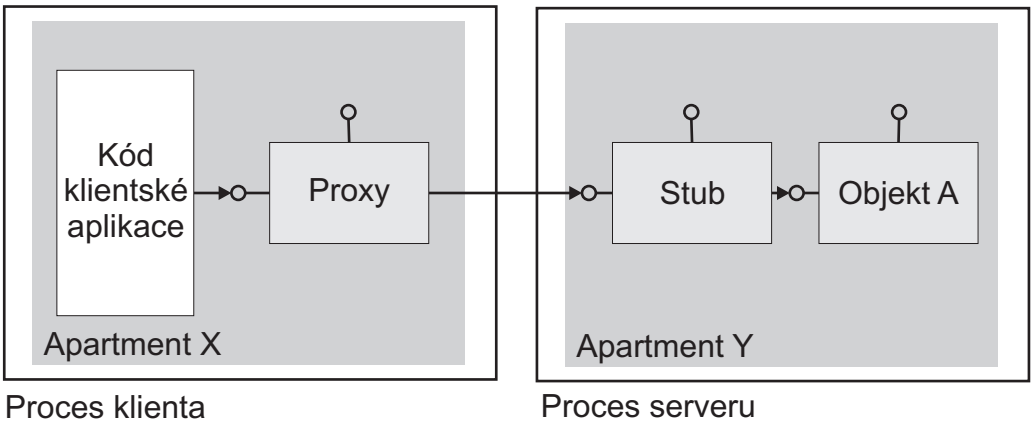
3.5.2 Marshalování parametrů

Při marshalování parametrů je spojení klienta a serveru zajišťováno pomocí dvou mezičlánků - *proxy* a *stub*. Proxy zastupuje instanci komponenty na straně klienta a stub je prostředníkem na straně instance komponenty. Klient pak ve skutečnosti místo volání metod rozhraní komponenty, volá metody rozhraní proxy, které je předává objektu stub. Ten tyto klientské požadavky realizuje voláním metod skutečné instance komponenty.

V závislosti na typu přenášených parametrů máme 3 možnosti jak je převést do nezávislé přenositelné podoby, neboli jak realizovat marshalování

- marshalování typovou knihovnou (*Type Library Marshaling*) - využívá knihovnu *oleaut32.dll*, která je standardní součástí operačního systému. Tato knihovna obsahuje potřebný marshalovací kód pro objekty proxy a stub. Podporuje ale pouze parametry které jsou OLE Automation kompatibilní, viz tab. 3.4.

- standardní marshalování (*Standard Marshaling*) - marshalovací knihovna je vytvořena ze souborů generovaných MIDL kompilátorem. Podporuje všechny typy



Obrázek 3.8: Spojení klienta a out-of-process serveru

parametrů nadefinovaných v IDL souboru.

- uživatelsky definované marshalování (*Custom Marshalling*) - marshalovací knihovna je vytvořena celá uživatelem. Umožňuje kontrolovat celý proces marshalování, ale je složitá na implementaci.

Typ IDL	Typ C++
boolean	bool
double	double
float	float
signed int	int(long)
signed short	short
enum	enum
BSTR	BSTR
CY	CY
DATE	DATE
IDispatch*	IDispatch*
IUnknown*	IUnknown*
SAFEARRAY	SAFEARRAY
VARIANT	VARIANT

Tabulka 3.4: Seznam OLE Automation kompatibilních typů

3.6 Registrace komponent

Před tím, než lze používat jakékoliv metody komponent uvnitř serveru, je nutné je zaregistrovat komponenty ve Windows Registry. In-process servery se registrují programem *regsvr32.exe*, přes exportované funkce *DllRegisterServer* a *DllUnregisterServer*. Registrace komponent out-of-process serveru se většinou provádí automaticky spuštěním serveru se specifickým parametrem. Server musí zaregistrovat své komponenty a marshalovací knihovny jednotlivých rozhraní, které komponenty implementují. Typovou knihovnu je nutno zaregistrovat také, pokud mají být komponenty využívány i klienty z jiných programovacích jazyků než C/C++, nebo používáme-li marshalování s typovou knihovnou. Informace lze zapsat do registru ručně, vytvořením dávkového souboru *.reg* a následným spuštěním tohoto souboru programem *regedit.exe*, nebo pomocí systémových funkcí přímo v kódu serveru. Postup jak správně registrovat komponenty obsahuje lit. [1].

3.7 Řízení životnosti COM serveru

3.7.1 Class Object

Vytvořit instanci komponenty lze více způsoby (Monikery, ClassObject,...). Všechny z nich jsou popsány v lit. [1]. *ClassObject* je obyčejná komponenta, která má za úkol pouze vytvářet instance komponenty s požadovaným rozhraním. Má podobnou funkci jako operátor *new* v C++. Důvodem, proč tento operátor nejde použít přímo, je skutečnost, že klientská aplikace aplikace neví nic o konkrétní implementaci komponenty. Každá komponenta musí mít svůj vlastní (jiný) ClassObject. ClassObject může implementovat libovolné uživatelské rozhraní sloužící k vytváření komponent, ale obvykle implementuje některé z již hotových rozhraní, připravených firmou Microsoft, např. *IClassFactory*. Toto rozhraní také využívá server vytvořený v této práci.

Rozhraní IClassFactory obsahuje kromě metod povinně odvozených z IUnknown také dvě další, viz tab. 3.5.

Atribut	Význam
CreateInstance	Vytvoří instanci komponenty s požadovaným rozhraním.Vrátí ukazatel na toto rozhraní.
LockServer	Uzamyká nebo odemyká ClassObject (a tedy i proces serveru) v paměti. Pokud je ClassObject uzamčen nemůže být z paměti uvolněn. Klient který uzamkl server je zodpovědný za jeho odemčení.

Tabulka 3.5: Seznam metod rozhraní IClassFactory

Jelikož je ClassObject také komponentou, musí být identifikován pomocí CLSID. Pro některé funkce volané klientem, který požaduje vytvořit instanci komponenty a získat ukazatel na rozhraní, je nutné, aby měl ClassObject stejné CLSID jako komponenta.

3.7.2 Založení instance komponenty klientem

Jestliže klientská aplikace potřebuje vytvořit instanci komponenty a tím spustit server, může to opět udělat více způsoby, viz lit. [1]. Používáme-li aktivaci přes ClassObject, klient musí volat jednu ze dvou funkcí Win API z tab. 3.6.

Atribut	Význam
CoCreateInstance	Vyvolá sled operací na straně serveru, nutných pro vytvoření instance komponenty. Tato funkce vyžaduje a automaticky pracuje s rozhraním IClassFactory. Vrátí ukazatel na instanci komponenty.
CoGetClassObject	Oproti předchozí funkci vytvoří pouze instanci ClassObject. Klient pak musí ještě použít funkce ClassObject k vytvoření instance komponenty. Použití IClassFactory není nutné.

Tabulka 3.6: Metody klienta pro založení komponenty

Obě tyto funkce vyžadují aby byla již instance ClassObject vytvořena v paměti. U in-process serveru se toto zajistí automatickým voláním exportované DLL funkce *DllGetClassObject* během operace CoCreateInstance, resp. CoGetClassObject. Out-of-process server musí pro každou komponentu vytvořit instanci ClassObject sám a pak ji

zaregistrovat do speciálního systémového seznamu pomocí funkce *CoRegisterClassObject*. Pro odstranění instance ClassObject ze seznamu server využije funkci *CoRevokeClassObject*.

In-process server (DLL, OCX, AX...soubor) může být odstraněn z paměti automaticky použitím dvou operací

- *CoUninitialize*
- *CoFreeUnusedLibraries*

Obě tyto metody interně prověří pomocí exportované DLL funkce *DllCanUnloadNow*, zda některá z instancí ClassObjectů serveru není uzamčena v paměti. Out-of-process server musí sám kontrolovat, kdy se může ukončit. Opět lze použít zaregistrovanou událost. Detailní popis řízení životnosti lze opět najít v lit. [1].

3.8 Connection Points

V případě, že nestačí komunikace ve směru od klienta k serveru a je potřeba realizovat obousměrnou komunikaci, můžeme využít speciálních rozhraní, vytvořených pro tento účel firmou Microsoft. Jsou to dvě povinná rozhraní

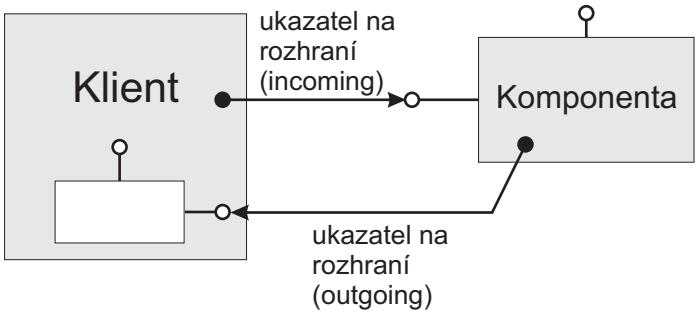
- *IConnectionPoint*
- *IConnectionPointContainer*

a dvě nepovinná

- *IEnumConnectionPoints*
- *IEnumConnections*

Implementuje-li komponenta tyto rozhraní, říká se jí také *Connectable Object*. Voláním metod těchto rozhraní si klient u komponenty může zaregistrovat svá rozhraní (tzv. *outgoing* rozhraní), jejichž metody pak může komponenta využívat. Situaci ilustruje obr. 3.9.

Detailní popis Connection Points lze najít v lit. [11].



Obrázek 3.9: Obousměrné spojení klienta s komponentou

3.9 Alokace paměti

Aby přenášení a marshalování parametrů mezi klientem a serverem probíhalo korektně a nedocházelo k únikům paměti, existují pro něj určitá pravidla.

Typ IDL	Alokace paměti
[in]	Klient předá hodnotu parametru, pro který alokoval paměť, server tuto hodnotu může jenom číst.
[in,out]*	Klient předá adresu parametru pro který alokoval paměť, server může hodnotu parametru číst i měnit.
[out]**	Použije se v případě že velikost parametru (např. pole) není předem známa. Klient alokuje ukazatel na paměť a předá adresu tohoto ukazatele serveru. Server alokuje paměť a uloží adresu paměti do ukazatele.
[in,out]**	Použije se v případě že se paměťová velikost parametru mění. Klient alokuje paměť, předá ukazatel na adresu alokovaného pole, server přečte nebo změní data a paměť podle potřeby dealokuje, resp. realokuje. Pak vrátí zpět ukazatel na adresu tohoto pole.

Tabulka 3.7: Alokace paměti podle typu parametru

Celou práci z pamětí určují dva hlavní parametry

- směr komunikace, tedy volá-li klient metody komponenty (volání incoming rozhraní), nebo volá-li komponenta metody klienta (volání outgoing rozhraní).

- typ parametru, tedy jedná-li se o vstupní (*in*), výstupní (*out*), nebo vstupně-výstupní (*in,out*).

Detailnější popis alokace dává tab. 3.7. V posledních dvou řádcích tab. 3.7 může velikost klientem alokované paměti měnit server. Aby tento proces alokace, dealokace či realokace fungoval správně, musí klient/server užívat speciálních COM funkcí určených pro správu paměti, které popisuje tab. 3.8. Jejich detailnější popis lze najít v lit. [12].

Metoda	Popis
CoTaskMemAlloc	Alokuje pamět dané velikosti.
CoTaskMemReAlloc	Změní velikost alokované paměti. Interně podle potřeby může volat CoTaskMemAlloc i CoTaskMemFree.
CoTaskMemFree	Dealokuje paměť.

Tabulka 3.8: COM funkce pro správu paměti.

Při volání metod outgoing rozhraní se v tab. 3.7 pozice klienta a serveru obrátí.

Kapitola 4

Standard OPC

OPC (*Ole for Process Control*) je skupina specifikací udávající předepsanou formu meziprocesní komunikace. Všechny OPC specifikace využívají model COM. Každá specifikace byla vytvořena pro jiný účel a má definované svoje COM komponenty a rozhraní. O tvorbu a rozvoj specifikací se stará organizace OPC Foundation. Sdružuje mnoho firem zabývajících se tvorbou OPC produktů. Jak už vypovídá jeho standardu, našel široké uplatnění v řízení technologických procesů. Aplikace vytvořené v rámci této práce využívají specifikaci OPC Data Access.

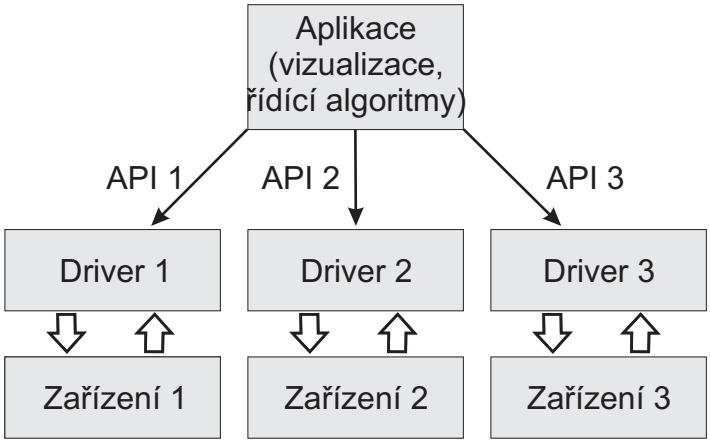
4.1 Úvod

Data Access byla první specifikací OPC standardu. Původně mělo zůstat pouze u ní. Je primárně navržena pro výměnu dat se zařízeními připojenými k počítači. Používá se přímo pro real-time výměnu dat ze senzorů, akčních členů, dat pro řídící algoritmy, operátorská stanoviště, či záznam dat do databází. Nejnovější verze této specifikace je OPC Data Access 3.0. Aplikace COM serveru a klienta vytvořené v této práci podporují OPC Data Access 2.05.

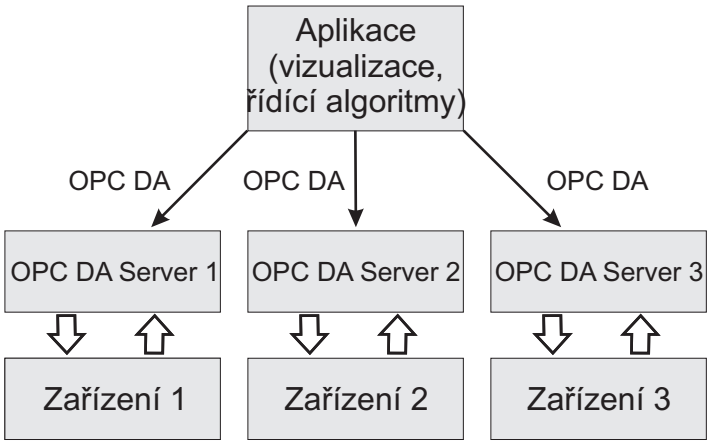
Snahou specifikace Data Access a koneckonců všech OPC specifikací je zjednodušení vývoje řídících systémů. Chce-li aplikace přistupovat k fyzickému zařízení, jde na to obvykle přes jeho ovladač. Podporuje pak aktuální verzi ovladače, při změně jeho verze je potřeba aplikaci přepsat. Chce-li aplikace najednou přistupovat k více zařízením, musí umět komunikovat s každým ovladačem. Jelikož různí výrobci většinou používají různá API pro svoje ovladače, vývoj takovéto aplikace je značně obtížný. Situaci ilustruje obr.

4.1.

V případě použití OPC se situace mění. Aplikace (OPC klient) bude přistupovat ke všem zařízením použitím stejného API daného OPC stanardem přes mezičlánky - OPC servery. OPC server je pak tím, kdo přistupuje k zařízení pomocí ovladače. V případě nekompatibility je pak nutno přepsat jenom část kódu serveru. V dnešní době většina výrobců hardware nabízí ke svým produktům také OPC servery. Novou situaci ilustruje obr. 4.2. Další výhodou OPC je možnost jednoduché organizace dat do různých skupin, logicky vztahujících se k technologii řízeného procesu.



Obrázek 4.1: Model klasické komunikace se zařízením



Obrázek 4.2: Model komunikace s využitím OPC

4.2 Pozadí COM v OPC Data Access

4.2.1 Typová knihovna a marshalování parametrů

Marshalování je v OPC Data Access prováděno pomocí standardní marshalovací knihovny *opcproxy.dll* obsahující proxy/stub kód. Tento soubor obsahuje také typovou knihovnu. Opcproxy.dll obsahuje deklarace všech rozhraní mimo IOPCCommon, resp. IOPCShutdown, viz podkap. 4.3.1.2, resp. 4.4.0.2. Tyto dvě rozhraní jsou sdíleny všemi OPC specifikacemi proto je kód pro jejich proxy/stub a typovou knihovnu umístěn do jiného souboru, aby nemusel být kopírován do knihoven pro každou specifikaci zvlášť. Tímto souborem je *opccomn_ps.dll*.

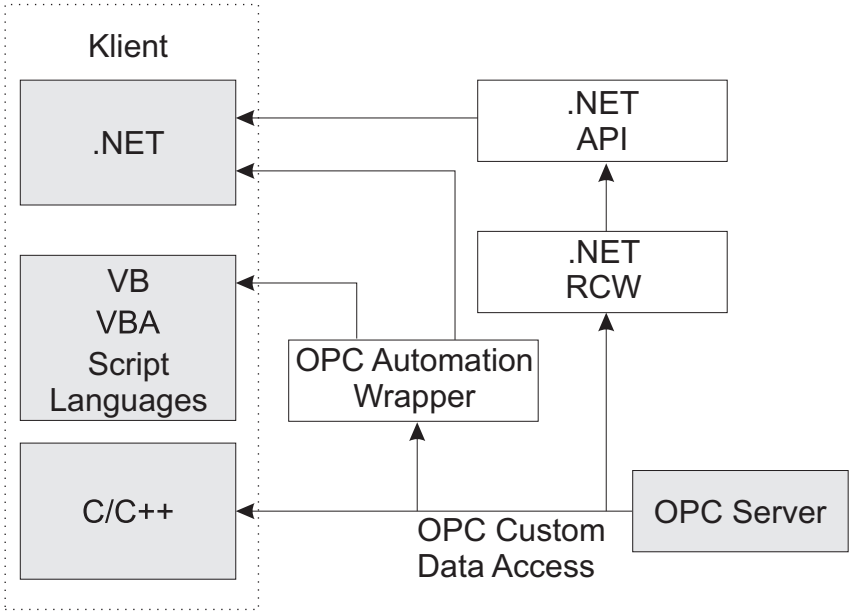
4.2.2 Automation

Deklarace v knihovnách v předešlé podkapitole podporují pro komunikaci s klientem pouze custom rozhraní. OPC servery můžou také volitelně implementovat podporu automation rozhraní. Daleko jednodušší je neimplementovat automation přímo v kódu serveru, tj. nerealizovat duální rozhraní, ale použít obálku. Touto obálkou je knihovna *opcdaauto.dll*, která automaticky převede custom rozhraní na automation. Při vývoji klienta v programovacím jazyce, kde potřebujeme automation rozhraní, pak stačí přidat tuto knihovnu do referencí.

Poslední verze opcdaauto.dll může být použita i v klientských aplikacích běžících na platformě .NET Framework. Z důvodů nutnosti zpětné kompatibility z předešlými verzemi, ale nevyužívá plně předností .NETu. Proto byla vyvinuta nová RCW knihovna, viz. 5.2, která zajišťuje zapouzdření COM rozhraní (*opcrcw.da.dll*), určená a optimalizovaná právě pro .NET Framework. I použití této obálky ale vyžaduje značné úsilí programátora při vývoji klienta/serveru. Programátor může ale s výhodou použít ještě .NET API, poskytované organizací OPC Foundation. Toto API tvoří jakousi „obálku obálky“, umožňující snadnou implementaci klienta/serveru. Situace je na obr. 4.3.

4.3 COM server pro OPC Data Access

COM server se podle standardu OPC skládá ze tří základních tříd, dodržujících určitou hierarchii, viz př. 4.1. Nejnižší třídou je OPC položka *(OPCItem)*. OPC položka

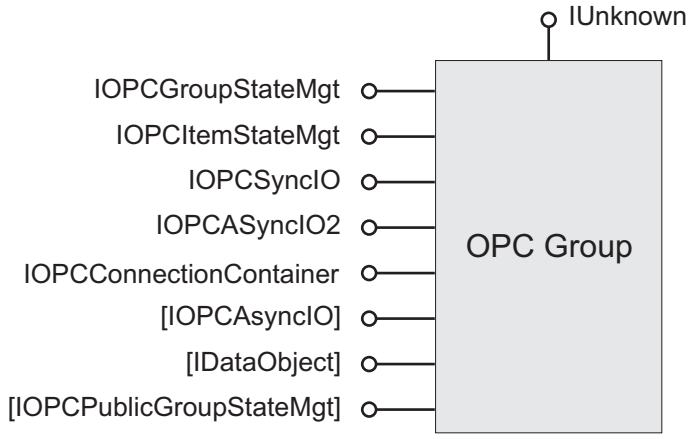


Obrázek 4.3: Obálky pro OPC klienty různých programovacích jazyků

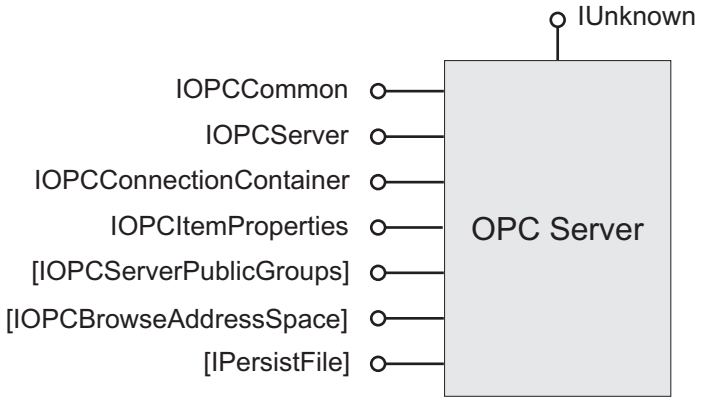
je zdrojem dat pro OPC klienty, právě nad ní provádějí operace čtení a zápis. Každá OPC položka je interně mapovaná na části adresního prostoru serveru (*Address Space*), tzv. *tagy*. Tagy mohou být sdíleny mezi všemi položkami všech OPC skupin a OPC serverů a jsou na ně kladeny největší nároky ve smyslu více-vláknové bezpečnosti (pro MTA apartment). Pomocí funkcí ovladačů konkrétního zařízení server podle potřeby aktualizuje adresní prostor. Bližší popis OPC položek je v podkap. 4.3.3. Implementace adresního prostoru není nijak omezena OPC standardem a závisí výhradně na programátorovi. Popis implementace adresního prostoru v konkrétní aplikaci je v podkap. 6.1.

OPC položky jsou organizovány do OPC skupin *(OPC Group)*. OPC skupina je již komponentou, s povinností implementovat rozhraní daná OPC standardem, viz obr. 4.4. Rozhraní uvedená v hranatých závorkách jsou podle specifikace OPC Data Access 2.05 nepovinná. Skupina umožňuje zakládat a rušit položky, obsahuje základní metody pro čtení/zápis do položek a globální nastavení atributů položek, viz podkap. 4.3.2.

Za ClassObject pro OPC skupiny by se dala považovat komponenta OPC server. Zakládá a ruší skupiny, nastavuje způsob čtení dat položek ve skupinách. Je také první komponentou jejíž instance vytváří klient po připojení k serveru a poskytuje základní informace pro propojení položek s adresním prostorem. Celá hierarchie je většinou logicky uspořádaná podle řízené technologie, viz př. 4.1.



Obrázek 4.4: Komponenta OPC Group



Obrázek 4.5: Komponenta OPC Server

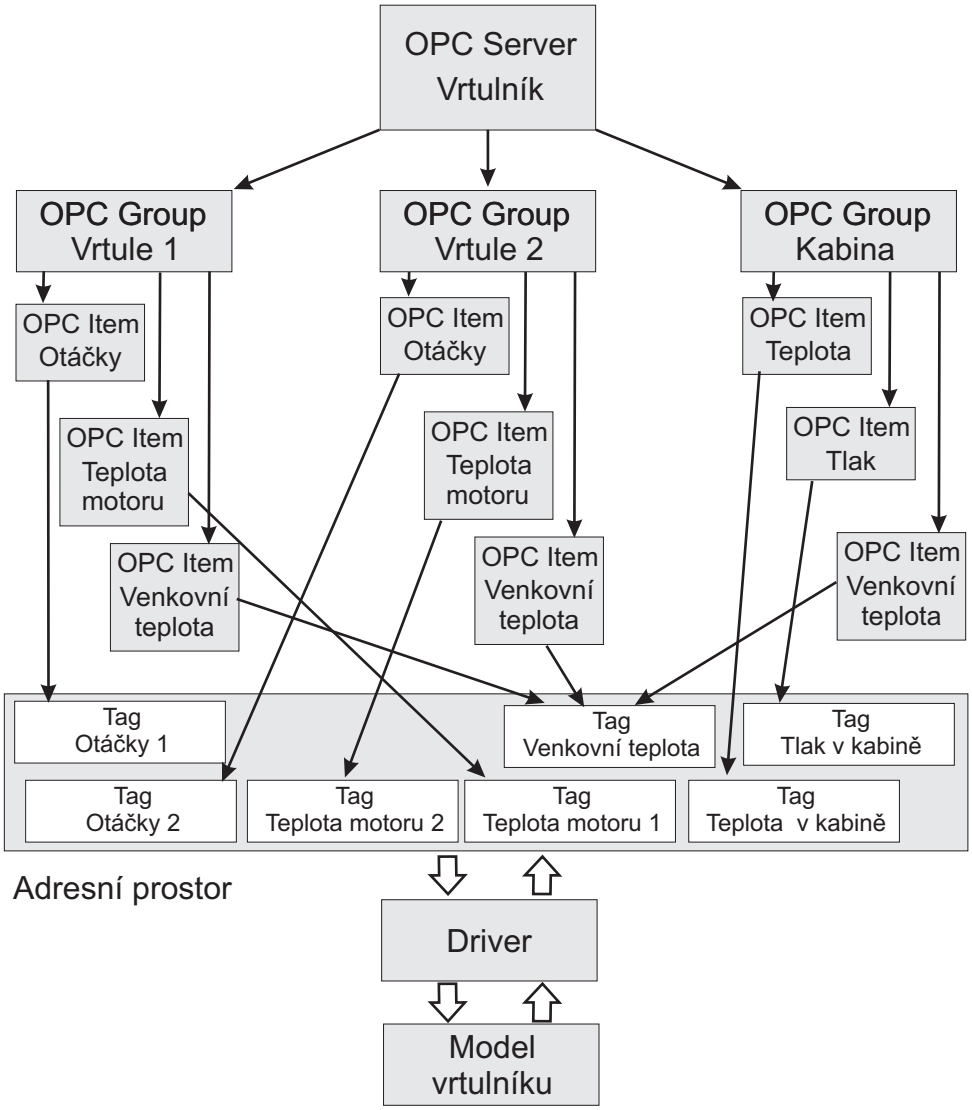
V dalších kapitolách jsou popsány jednotlivá rozhraní komponent implementovaná v konkrétní aplikaci COM OPC serveru. Detailnější popis jednotlivých metod lze najít v lit. [6].

4.3.1 Komponenta OPC Server

Každá instance komponenty OPC server má tyto základní atributy

- název výrobce a produktu (*VendorInfo*).
- akt. čas systému (aktualizuje pouze na dotaz) (*Current Time*).

Příklad 4.1 (Uspořádání OPC Serveru pro řízení lab. modelu vrtulníku):



Obrázek 4.6: Příklad uspořádání OPC serveru

- čas poslední aktualizace některé z položek (*Last Update Time*).
- aktuální status serveru (spuštěn, stojí, apod.) (*Server State*).
- počet založených skupin (*Group Count*).
- zatížení serveru (v %) (*Band Width*).
- číslo verze, podverze a sestavení (*Major Version, Minor Version, Build Number*).

Základní nastavení atributů se provede při vytvoření instance komponenty.

4.3.1.1 IOPCServer

Rozhraní IOPCServer je hlavním rozhraním komponenty OPC Server. Obsahuje metody k manipulaci se skupinami a prohlížení adresního prostoru. Každá skupina musí mít v rámci jedné instance komponenty OPC server unikátní jméno.

Metoda	Popis Metody
AddGroup	Založí OPC skupinu a nastaví její základní atributy, viz podkap. 4.3.2.1.
GetErrorString	Vrátí textový řetězec pro daný kód HRESULT. Závisí na lokalizaci serveru, viz podkap. 4.3.1.2.
GetGroupByName	Vrátí ukazatel na skupinu podle daného jména.
GetStatus	Vrátí základní atributy serveru, viz podkap. 4.3.1.
RemoveGroup	Zruší skupinu.
CreateGroupEnumerator	Vytvoří objekt implementující speciální rozhraní IEnumUnknown, viz lit. [15], které umožňuje jednoduchý průchod skupinami vytvořenými v OPC serveru.

Tabulka 4.1: Seznam metod rozhraní IOPCServer

4.3.1.2 IOPCCommon

Toto rozhraní umožňuje lokalizaci serveru. Tím se myslí nastavení „mateřského jazyka“, kterým bude server komunikovat s klientem v určitých metodách. Každý jazyk je identifikován pomocí unikátního čísla, tzv. LCID (*LocaleID*).

Metoda	Popis Metody
SetLocaleID	Nastaví jazyk pro komunikaci.
GetLocaleID	Vrátí aktuální jazyk komunikace.
QueryAvailableLocaleIDs	Vrátí všechny podporované jazyky.
GetErrorString	Má úplně stejnou funkci jako stejnojmenná metoda rozhraní IOPCServer.
SetClientName	Nastaví jméno připojeného klienta.

Tabulka 4.2: Seznam metod rozhraní IOPCCommon

4.3.1.3 IOPCBrowseServerAddressSpace

Toto rozhraní umožňuje klientovi procházet adresní prostor serveru. Používají se základní dva typy adresního prostoru

- plochý (*flat*) - adresní prostor není organizován do skupin.
- větvený (*leaf*) - adresní prostor je organizován do skupin. Každá skupina tagů zastupuje obvykle jednu charakteristickou veličinu řízeného systému (z pohledu serveru jeden z tagů). Prvky skupiny pak bývají atributy veličiny, jako je např. horní/dolní limit, rozsah hodnot, hysterezi apod. Z pohledu OPC serveru jsou to také tagy, jen hierarchicky organizovány. Každý tento atribut má v rámci skupiny tagů unikátní jméno. Klient může získat vysvětlení, co který z těchto parametrů znamená, pomocí volání metod rozhraní IOPCItemProperties.

Server nemusí poskytovat hned plná jména tagů, místo toho může klientům posílat jména neúplná (*hints*) a poskytnout plné jméno až po dotazu na konkrétní tag .

Metoda	Popis Metody
QueryOrganization	Vrátí typ adresního prostoru.
ChangeBrowsePosition	Umožňuje procházet větveným prostorem.
BrowseOPCItemIDs	Vytvoří objekt se speciálním rozhraní IEnumString, viz lit. [16], které dokáže procházet seznamem tagů. Tento seznam může být filtrovaný.
GetItemID	Vrátí úplný název tagu, vstupem je hint.
BrowseAccessPaths	Vrátí seznam (IEnumString) možných cest k tagu v adresním prostoru.

Tabulka 4.3: Seznam metod rozhraní IOPCBrowseServerAddressSpace

4.3.1.4 IOPCItemProperties

Každá OPC položka obsahuje nějaké povinné atributy, viz seznam v podkap. 4.3.3. Metody rozhraní IOPCItemProperties pro každou položku automaticky vrací z těchto povinných atributů prvních šest. OPC položka může obsahovat i další nepovinné parametry např. horní/dolní limit, rozsah hodnot, hysterezi apod. Ty můžou být statické

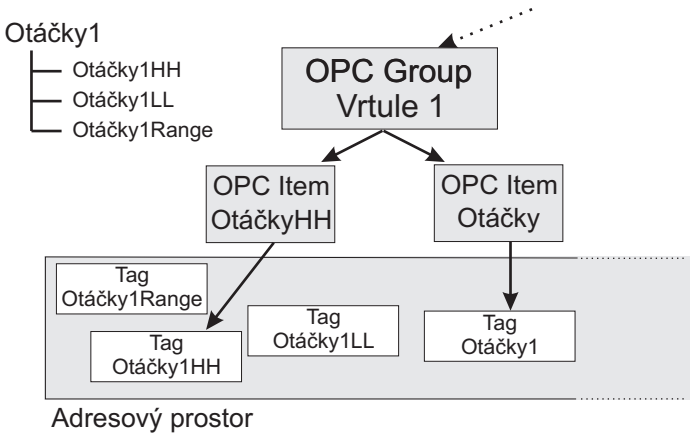
s pevně nadefinovanými hodnotami, ale většinou existují jako další tagy v adresním prostoru. Rozhraní IOPCItemProperties pak vrací jako další (nepovinné) atributy v seznamu nadefinované statické hodnoty či akt. hodnoty těchto tagů.

Každý atribut je identifikován pomocí čísla (*ID*) definovaného OPC specifikací, viz lit. [6]. Ke každému atributu je také v seznamu uveden textový popis.

Metoda	Popis Metody
QueryAvailableProperties	Vrátí seznam ID dostupných atributů k dané OPC položce.
GetItemProperties	Vrací textové popisy a aktuální hodnoty pro daný seznam ID a danou OPC položku.
LookupItemIDs	Vrací plné názvy tagů pro daný seznam ID nepovinných atributů a danou OPC položku, pokud tyto tagy existují v adresním prostoru.

Tabulka 4.4: Seznam metod rozhraní IOPCItemProperties

Příklad 4.2 (Příklad struktury adresního prostoru):



Obrázek 4.7: Příklad struktury adresního prostoru

Na obr. 4.7 je podrobněji zobrazena část adresového prostoru z př. 4.1. Tagy jsou logicky uspořádány, adresový prostor má větvenou strukturu. Pomocí metod rozhraní IOPCBrowseAddressSpace může klient získat jména všech tagů v adresovém prostoru a posléze na ně namapovat nějakou položku skupiny. Udělal to již pro tag Otáčky1

a Otáčky1HH. Pro tyto položky bude mít volání metod IOPCItemProperties typicky odlišné chování. Pro položku OtáčkyHH vrátí pouze základní atributy. Pro položku Otáčky vrátí kromě základních atributů také hodnoty tagů Otáčky1HH, Otáčky1LL a Otáčky1Range, i když tyto tagy nefigurují ve skupině jako OPC položky.

4.3.1.5 IConnectionPointContainer a IConnectionPoint

Tyto dvě standardní COM rozhraní slouží OPC klientovi k povinné registraci outgoing rozhraní IOPCShutDown, viz podkap. 4.4.0.2.

4.3.2 Komponenta OPC Group

4.3.2.1 IOPCGroupStateMgt

Klientovi slouží k dodatečnému nastavení atributů OPC skupiny. Prvotní nastavení atributů OPC skupiny probíhá při jejím vytvoření pomocí metody AddGroup z rozhraní IOPCServer. Hlavní atributy OPC skupiny jsou

- jméno skupiny (*Name*) - musí být v rámci instance komponenty OPC server unikátní.
- obnovovací frekvence (*Update Rate*) - min. frekvence s jakou OPC server musí aktualizovat hodnoty položek ve skupině.
- pásmo necitlivosti (*Deadband*) - procentuální vyjádření min. platné změny. Pokud se hodnota tagu změní méně, OPC položka nemusí být aktualizována. Tento parametr je směrodatný pouze u analogových veličin.
- časový posun (*TimeBias*) - nastaví časový posun mezi klientskou aplikací a serverem. Používá se v případech, kdy klient a server operují v jiných časových zónách. Správný čas v časové zóně klientské aplikace = čas pořízení hodnoty položky (*Timestamp*) + Timebias.
- lokalizace (*LCID*) - říká v jakém jazyce má server vracet textové řetězce pro tuto skupinu. Toto nastavení se neprojeví do názvů položek, tagů ani skupin.
- příznak aktivity (*Active flag*) - říká zda je skupina aktivní/neaktivní. Tento příznak ovlivňuje chování serveru při čtení položek ve skupině, viz podkap. 4.3.2.4, 4.3.2.5, příp. 4.4.0.1.

- handle serveru (*ServerGroupHandle*) - unikátní číslo skupiny v rámci OPC serveru.
- handle klienta (*ClientGroupHandle*) - unikátní číslo skupiny v rámci klienta.

Metoda	Popis Metody
GetState	Zjistí hodnoty výše uvedených atributů.
SetState	Nastaví hodnoty výše uvedených atributů.
SetName	Nastaví nové jméno skupiny.
CloneGroup	Vytvoří novou skupinu se stejnými atributy (kromě jména,handle serveru, handle klienta) a stejnými položkami.

Tabulka 4.5: Seznam metod rozhraní IOPCGroupStateMgt

Metoda	Popis Metody
AddItems	Vytvoří vybrané položky, nastaví těmto položkám základní hodnoty atributů.
ValidateItems	Zkontroluje, zda vložení vybraných položek nezpůsobí chybu. Zpravidla je tato metoda volána před AddItems.
RemoveItems	Odebere vybrané položky.
SetActiveState	Nastaví vybraným položkám příznak aktivity.
SetClientHandles	Nastaví vybraným položkám unikátní číslo pro identifikaci v klientské aplikaci.
SetDataTypes	Nastaví datový podtyp vybraným položkám. Datový typ je vždy VARIANT, viz lit. [17]. SetDataTypes nastaví pouze typ informace který je ve struktuře VARIANT obsažen.
CreateEnumerator	Vytvoří objekt implementující speciální rozhraní IEnumOPCItemAttributes, viz lit. [6], které umožňuje jednoduchý průchod položkami vytvořenými ve skupině.

Tabulka 4.6: Seznam metod rozhraní IOPCItemMgt

4.3.2.2 IOPCItemMgt

Toto rozhraní je hlavním rozhraním skupiny pro práci s OPC položkami. Každá OPC položka nemusí mít v rámci skupiny unikátní jméno. Je jednoznačně identifikována pomocí unikátního čísla, které generuje server i klient při založení položky (*ServerHandle, ClientHandle*).

4.3.2.3 IConnectionPointContainer a IConnectionPoint

V případě OPC skupiny slouží tyto rozhraní k možnosti registrace outgoing rozhraní IOPCDataCallback. Metody rozhraní IOPCDataCallback a popis jsou v podkap. 4.4.0.1.

4.3.2.4 IOPCSyncIO

IOPCSyncIO je jedno z rozhraní OPC skupiny sloužící ke čtení/zápisu dat do položek. Čtení a zápis je prováděn synchronně, tzn. vlákno volající metody tohoto rozhraní čeká na dokončení operace serverem.

Metoda	Popis Metody
Read	Přečte hodnoty z tagů do vybraných položek. Klient může požadovat současnou aktualizaci tagů ze zařízení. Při tomto požadavku nezáleží zda je položky a skupina aktivní. Není-li tomu tak, operace se provede pouze pro aktivní položky v aktivní skupině.
Write	Pošle data z vybraných položek na obsluhované zařízení.

Tabulka 4.7: Seznam metod rozhraní IOPCSyncIO

4.3.2.5 IOPCASyncIO2

Narozdíl od IOPCSyncIO, jsou metody tohoto rozhraní vykonávány asynchronně. Vlákno, které započne provádět asynchronní operaci, nečeká na její dokončení. Když je operace ukončena, server použije metody outgoing rozhraní IOPCDataCallBack k obeznámení klienta o výsledcích operace. Pokud operace ještě nebyla ukončena, může klient operaci zrušit.

Metoda	Popis Metody
Read	Přečte hodnoty z tagů do vybraných položek. Automaticky aktualizuje tagy hodnotami ze zařízení.
Write	Pošle data z vybraných položek na obsluhované zařízení.
Refresh2	Přečte hodnoty z tagů do všech aktivních položek aktivních skupin. Platí pouze pro tagy, které změnily hodnotu. Klient může požadovat současnou aktualizaci tagů ze zařízení.
Cancel2	Zruší nedokončenou operaci Write, Read nebo Refresh2.
SetEnable	Nastavuje chování automatické zpětné vazby.
GetEnable	Zjistí zda je automatické zpětné vazba povolena/zakázána.

Tabulka 4.8: Seznam metod rozhraní IOPCASyncIO2

4.3.3 OPC Item

OPC položka nemusí implementovat žádná předepsaná rozhraní. Každá OPC položka ale musí mít k dispozici atributy

- aktuální hodnota (*Current Value*).
- časová značka (*TimeStamp*) - čas poslední aktualizace položky.
- kvalita aktuální hodnoty (*Quality*) - hodnocení věrohodnosti aktuální hodnoty.
- přístupová práva (*AccessRights*) - definují možnosti čtení/zápisu do položky.
- původní datový typ (*Native Datatype*) - definuje typ informace v datové struktuře VARIANT, v níž je uložena aktuální hodnota
- max. obnovovací frekvence serveru (*Server Scan Rate*).
- unikátní číslo v rámci skupiny OPC serveru (*ServerHandle*).
- unikátní číslo v rámci skupiny klienta (*ClientHandle*).
- příznak aktivity (*Activity flag*) - říká, zda je položka aktivní/neaktivní, definuje chování serveru při čtení dat, viz podkap. 4.3.2.4, 4.3.2.5, příp. 4.4.0.1.

- požadovaný datový typ (*Requested Datatype*) - definuje typ informace v datové struktuře VARIANT pro aktuální hodnotu, jež vyžaduje klient. Musí být kompatibilní s původním typem informace (Native Datatype), jinak položka nejde založit/přečíst.
- doplňkový identifikátor tagu (*Blob*) - identifikátor, který umožní rychlejší zakládání a rušení položek. Je předán klientovi při vytvoření položky a klient ho pak může využít při zakládání/rušení dalších položek.

Prvních šest atributů z nich je automaticky předáváno klientovi při volání metod rozhraní IOPCItemProperties.

4.4 COM klient pro OPC Data Access

Klientská aplikace podle potřeby zakládá instance komponent a ostatních objektů serveru a využívá metody rozhraní. Povinností klienta je implementace outgoing rozhraní IOPCShutDown a IOPCDataCallback.

4.4.0.1 IOPCDataCallback

Server volá metody rozhraní IOPCDataCallback při dokončení asynchronních operací, viz tab. 4.8. Používá se také pro automatickou aktualizaci a obeznámení klienta o změně aktuální hodnoty či kvality položky ve skupině (automatickou zpětnou vazbu).

Metoda	Popis Metody
OnWriteComplete	Pošle výsledky operace IOPCASyncIO2::Read.
OnReadComplete	Pošle výsledky operace IOPCASyncIO2::Write.
OnCancelComplete	Pošle výsledky operace IOPCASyncIO2::Write.
OnDataChange	Slouží k automatickému zpětné vazbě, nebo jako reakce na operaci IOPCASyncIO2::Refresh2. Aby klient rozlišil proč byla tato metoda volána, server pří reakci na Refresh2 nastaví jeden z parametrů metody na smluvenou hodnotu.

Tabulka 4.9: Seznam metod rozhraní IOPCDataCallback

Pokud klient požaduje automatické obeznámení při změně dat položky musí podniknout následující kroky

1. Zaregistrovat rozhraní IOPCDataCallback.
2. Zapnout automatickou zpětnou vazbu pomocí volání IOPCAsyncIO2::SetEnable.
3. Aktivovat skupiny, jež obsahují položky pro které má automatická zpětná vazba fungovat.
4. Aktivovat položky pro které má zpětná vazba fungovat.

4.4.0.2 IOPCShutdown

Obsahuje kromě metod IUnknown pouze jednu důležitou metodu - *ShutdownRequest*. Server volá tuto metodu v případě, že si přeje odpojení klienta. Klient pak musí povinně uvolnit veškeré své založené instance komponent.

Kapitola 5

Interoperabilita s prostředím .NET

Pro aplikace běžící na platformě .NET Framework jsou vytvořeny již hotové třídy, které slouží k práci se systémovými prostředky. Programátorovi při použití těchto tříd odpadá spousta práce, protože platforma .NET se sama stará o bezpečné a optimalizované spouštění kódu. Interně pak .NET Framework využívá standardního API operačního systému.

Jsou ale situace, kdy v knihovně tříd .NET Framework ekvivalentní funkcionalitu nenalezneme a potřebujeme využít původních tříd přímo. Pro přímé propojení .NET Framework a nespravovaných aplikací či knihoven tříd slouží tzv. *obálky* (*wrappers*). Obálky si lze představit jako třídy, které zapouzdřují původní třídy. Aplikace .NET pak volají metody tříd obálky, kterým rozumí, a obálka překládá tyto požadavky a jejich parametry na volání původních metod. Detailnější popis podkapitol lze najít v lit. [2].

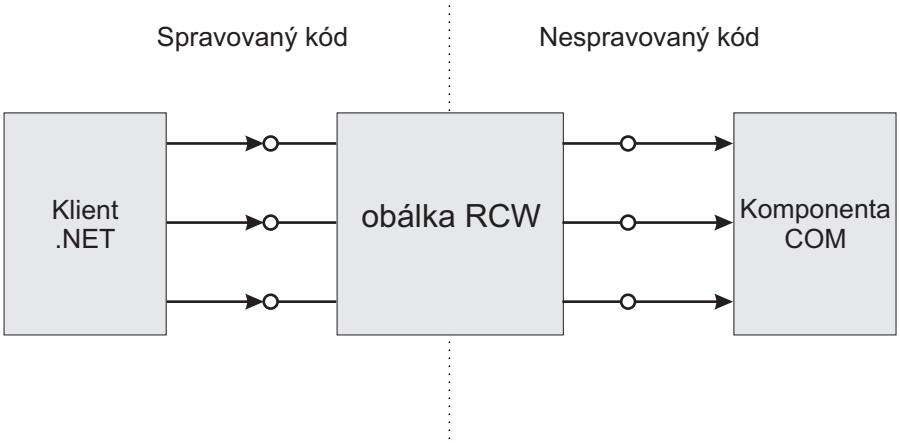
5.1 Funkce uložené ve Win32 DLL

Funkce uložené v nespravovaných DLL knihovnách (ať už systémových či uživatelských) mohou být zpřístupněny pomocí služby *Platform Invocation Services* (*PInvoke*). Když volá kód nespravovanou funkci, načte služba PInvoke soubor DLL obsahující definici funkce do paměti, vyhledá adresu funkce a převede vstupní argumenty na typy používané funkcí v DLL knihovně. Potom aktivuje funkci a přesune řízení toku programu do nespravované knihovny DLL.

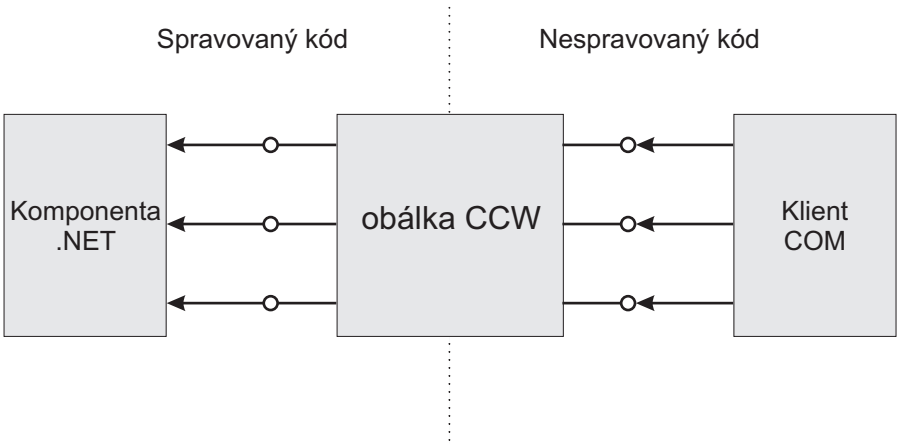
5.2 COM

Komponenty modelu COM nejsou s komponentami .NET přirozeně kompatibilní, protože jsou postaveny na odlišné interní architektuře. K vytvoření obálky je potřeba služeb *COM Interop Services*. Tato obálka překládá specifická volání vysílaná klientem běžícím ve spravovaném kódu na požadavky specifického volání metod komponenty modelu COM. Pro tyto obálky se používá zkratka RCW (*Runtime Callable Wrapper*). Tyto obálky lze jednoduše vytvářet pomocí nástroje prostředí .NET zvaného *Type Library Importer* (TlbImp.exe), který obálku RCW automaticky vytvoří z typové knihovny dané komponenty.

Podobně jako lze ve spravovaném kódu používat COM komponenty, lze rovněž komponenty objektového modelu .NET používat klienty běžícími ve standardním nespravovaném kódu. Tyto obálky jsou známy pod zkratkou CCW (*COM Callable Wrappers*). V tomto případě potřebujeme naopak typovou knihovnu vygenerovat. K tomuto účelu zde již opět existuje nástroj prostředí .NET, zvaný *Type Library Exporter* (TlbExp.exe).



Obrázek 5.1: Komunikace klienta prostředí .NET a komponenty COM



Obrázek 5.2: Komunikace COM klienta a komponenty prostředí .NET

Kapitola 6

Popis realizovaného OPC serveru

Postupy a modely zmíněné v předešlých kapitolách jsem aplikoval do praxe. Specifikaci OPC Data Access jsem využil pro výměnu dat s konkrétním zařízením. Tímto zařízením je karta Advantech 1750, využívaná k měření v průmyslových aplikacích. Karta je běžně dostupná a instaluje se do PCI slotu osobního počítače. Popis karty je uveden v příloze A. Dle zadání jsem vytvořil tři aplikace - OPC server, manažer OPC serveru a OPC klienta. Všechny aplikace jsou přiloženy na CD-ROM, viz příloha D. Následující kapitoly poskytují popis těchto aplikací. Detailnější popis lze nalézt přímo ve zdrojovém kódu aplikací.

Po vytvoření aplikací jsem je také testoval některými již hotovými nástroji. Přehled těchto testů je v příloze C.

6.1 Server

V aplikaci *AOPCSvr.exe* jsem implementoval funkce OPC serveru. Je to nejdůležitější aplikace vytvořená v této práci a může plně fungovat nezávisle na ostatních dvou. Server je konzolovou Win32 aplikací implementovanou v jazyce C++. Nevyužívá žádných speciálních knihoven typu MFC (*Microsoft Foundation Class*).

Bylo potřeba implementovat

- těla povinných funkcí modelu COM, viz kap. 3.
- těla povinných funkcí dané specifikací OPC Data Access 2.05, viz kap. 4.
- adresní prostor serveru.

- komunikaci s manažerem serveru pomocí pojmenovaných rour, viz podkap. 2.4.

Dále jsem k serveru přidal možnost záznamu chyb do textového souboru a načtení konfigurace serveru ze souboru formátu XML.

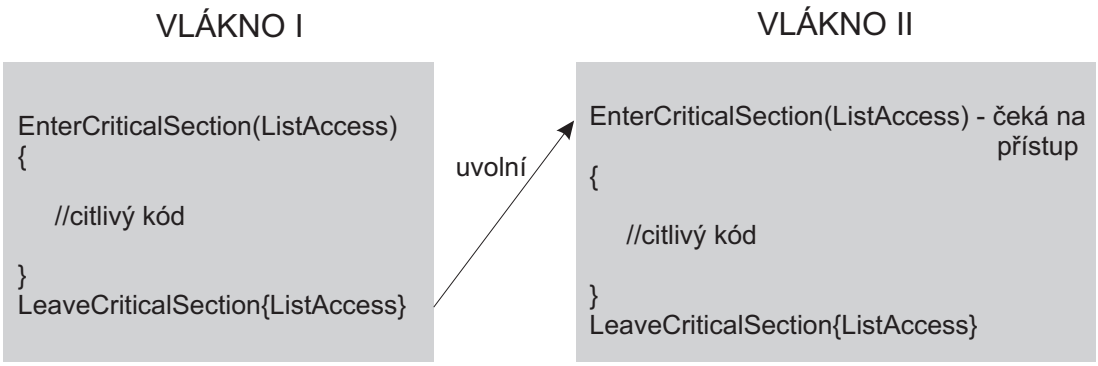
6.1.1 Parametry COM modelu

Aplikace funguje jako out-of-process server. Obsahuje více vláken, proto jsem volil MTA apartment. Všechna vlákna v rámci procesu serveru pak snadno přistupují k metodám rozhraní bez nutnosti marshalování. K marshalování dochází tedy jen při volání metod rozhraní klientskou aplikací (kterou je jiný proces), pomocí standardních marshalovacích knihoven, viz podap. 4.2.1.

Problémem MTA apartmentu je nutnost synchronizace přístupu jednotlivých vláken ke sdíleným prostředkům, jako je např. adresní prostor. V aplikaci serveru využívám dva typy synchronizačních objektů, a to kritické sekce a události. Detailní informace o synchronizačních objektech lze najít v lit. [18].

6.1.1.1 Kritické sekce

Příkladem využití kritické sekce může být přidání/odebrání tagu z adresního prostoru. Vlákno, které má v úmyslu vykonat kód pracující se seznamem tagů, nejdříve vstoupí do kritické sekce s citlivým kódem. Tím se tento kód stane dočasně nedostupný pro ostatní vlákna, která na přístup do kritické sekce musí čekat než ji vlákno uvnitř sekce znovu uvolní. Situace je na obr. 6.1.



Obrázek 6.1: Využití kritických sekcí

Při použití kritických sekcí je potřeba dbát určité opatrnosti. Může lehce dojít k totálnímu zablokování aplikace - *deadlocku*. K deadlocku dojde např. je-li vlákno uvnitř kritické sekce a znovu zažádá o přístup do ní. Ten ale nikdy nedostane, protože by kritickou sekci nejprve muselo uvolnit. Vlákno se tedy nadobro zablokuje.

6.1.1.2 Události

O událostech jsem se zmínil již v podkap. 2.4.1.3, kde jsou systémové události využity při neblokovacím režimu pojmenovaných rour. Dále události využívám k ukončování vláken. Všechna vlákna pracují ve smyčkách, kde na začátku dochází ke kontrole stavu dvou hlavních událostí. První z nich je signalizuje připojení klienta, který vytvořil instanci některé z komponent, druhá připojení manažera, který může konfigurovat server. Aplikace serveru se samovolně neukončí dříve než tyto události budou signalizovat zrušení poslední instance kterékoli komponenty. Současně musí být také odpojený manažer.

6.1.2 Hlavní části aplikace

Aplikace obsahuje několik základních tříd

- *COPCHead* - základní globální třída, spravující seznam připojených klientů, adresní prostor a konfiguraci globálních proměnných serveru.
- *CTag* - zastupuje tag, základní stavební jednotku adresního prostoru.
- *COPCClassFactory* - vytváří instance třídy COPCServer.
- *COPCServer* - implementuje rozhraní komponenty OPC Server. Obsahuje také další interní funkce které jsem implementoval k zajištění správné funkčnosti.
- *COPCGroup* - implementuje rozhraní komponenty OPC Group + další interní funkce.
- *COPCItem* - reprezentuje OPC položku.
- *CManagerIO* - implementuje komunikační protokol, viz podkap. 6.2.1.9 a zajišťuje komunikaci pomocí pojmenovaných rour.
- *CLog* - zajišťuje záznam chyb.

6.1.2.1 COPCHead (*OPCHead.h*)

Třída COPCHead je základní třídou celé aplikace. Její funkce lze rozdělit do několika skupin, podle druhu dat se kterými pracují, viz tab. 6.1.

Účel	Umístění
Metody zabezpečující práci se seznamem klientů (odpojení, připojení, omezení počtu), dále pak načtení konfigurace se souboru	ServerGlobal.cpp
Metody nastavující časovou konstantu pro automatickou aktualizaci tagů v adresním prostoru ze zařízení (karty Advantech 1750).	UpdateRate.cpp
Metody spravující adresní prostor (vytvoření/zrušení tagu, zjištění/nastavení vlastností tagu, vlákno pro automatickou aktualizaci tagů)	TagList.cpp
Metody zabezpečující vyřizování asynchronních požadavků (čtení/zápis) od všech připojených klientů	AsyncTransList.cpp
Metody zabezpečující inicializaci, čtení, zápis, nastavení přerušení na PCI kartě	Device.cpp

Tabulka 6.1: Skupiny metod třídy COPCHead

COPCHead automaticky zakládá dvě vlákna. První vlákno slouží k automatické aktualizaci tagů adresního prostoru. Vlákno periodicky opakuje aktualizace s danou časovou konstantou, následně plní funkci automatického obeznámení klienta o změnách položek. Časovou konstantu lze nastavit užitím funkcí z *UpdateRate.cpp*. Druhé vlákno automaticky vyřizuje asynchronní požadavky klientů.

Z důvodů jednoduchosti má třída a tím i celá aplikace následující omezení

- plochý adresní prostor, viz podkap. 4.3.1.3.
- počet tagů v adresním prostoru je max. 10.
- max. frekvence pro aktualizaci tagů je 10Hz, min. časová konstanta je tedy 100ms.
- max. počet připojitelných klientů je 11.

Tato čísla jsou postačující pro to, aby server vyhověl v testu kompatibility. Z max. frekvence lze usoudit že se server nehodí pro řízení velmi rychlých procesů a nevyužívá plně možnosti PCI karty, která umožňuje změnu hodnoty na vstupech/výstupech s frekvencí až 10Khz.

Adresní prostor je řešen jako pole o max. 10 prvcích. Při odebrání prvku jsou ostatní prvky pole přesunuty blíže k počátku pro rychlejší vyhledávání. Sejným způsobem je řešen seznam skupin třídy COPCServer nebo seznam položek pro třídu COPCGroup. Vedla mě k tomu zkušenost, že do těchto seznamů se příliš nepřidává/neodebírá, ale spíše se v nich vyhledává. Naproti tomu, seznam asynchronních požadavků je řešen jako řetězený seznam. Prvky se hlavně budou přidávat na jeho konec nebo odebírat z jeho začátku, vyhledávat se v něm bude jen zřídka.

Funkce skupiny z *Device.cpp* interně využívají funkcí ovladače karty, viz příloha A.5. Ten je dodáván spolu s kartou firmou Advantech v klasické DLL knihovně.

Konfigurace serveru, tedy max. počet připojitelných klientů, časová konstanta pro aktualizaci tagů apod., je uložena v souboru ve formátu XML. Pro načtení XML využívám již hotový XML *parser*. Parser je program, který rozčlení XML na jednotlivé položky. Autorem je Frank Vanden Berghen, viz lit. [21]. Jméno souboru je pevně nastaveno na *Config.xml*. Struktura konfiguračního souboru je na obr. 6.10.

6.1.2.1.1 Přerušení Pro řízení frekvence aktualizace adresního prostoru lze kromě časové konstanty využít i periodické přerušení karty. Zdroje přerušení jsou popsány v příloze A. Použití přerušení má dvě omezení

- možnost použití pouze interního zdroje přerušení karty.
- frekvence přerušení je nastavitelná od 10Hz do 0.002328Hz.

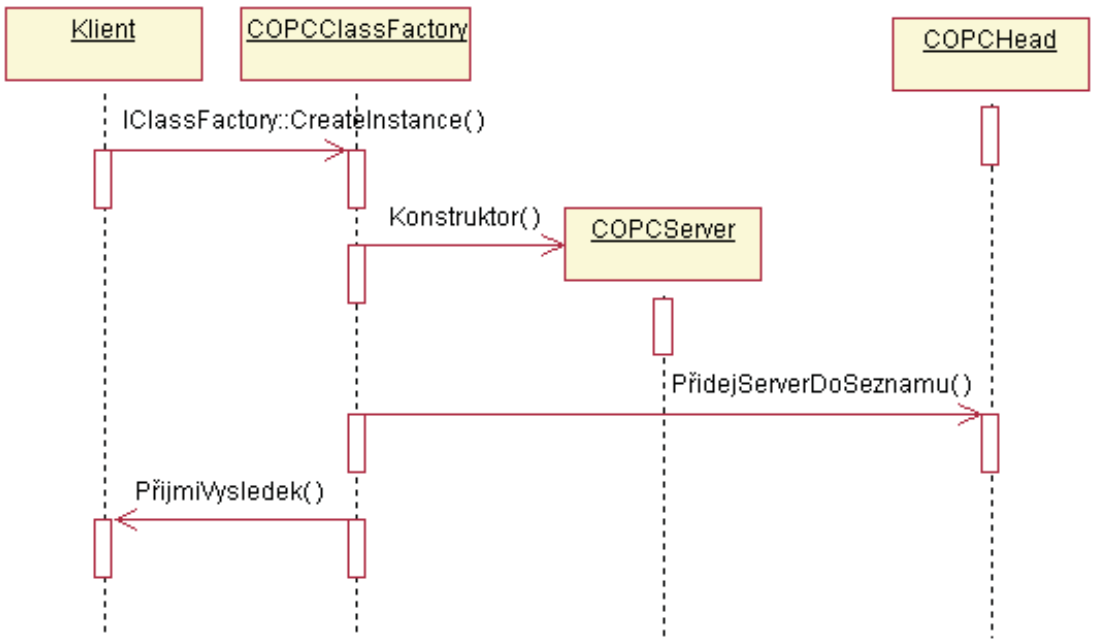
6.1.2.2 CTag (*Tag.h*)

Třída CTag reprezentuje tag z adresního prostoru serveru. Obsahuje pouze datové položky obdobné datovým položkám OPC Item, viz podkap. 4.3.3. O veškeré čtení/zápis se starají funkce třídy COPCHead.

6.1.2.3 COPCClassFactory (*OPCClassFactory.h*)

Třída COPCClassFactory implementuje rozhraní IClassFactory, viz podkap. 3.7.1. V této aplikaci slouží k vytváření instancí třídy COPCServer. Sled akcí při vytváření instance COPCServer je na obr. 6.2.

Příklad 6.1 (Založení komponenty OPC server):



Obrázek 6.2: Založení instance komponenty OPC server

6.1.2.4 COPCServer (*OPCServer.h*)

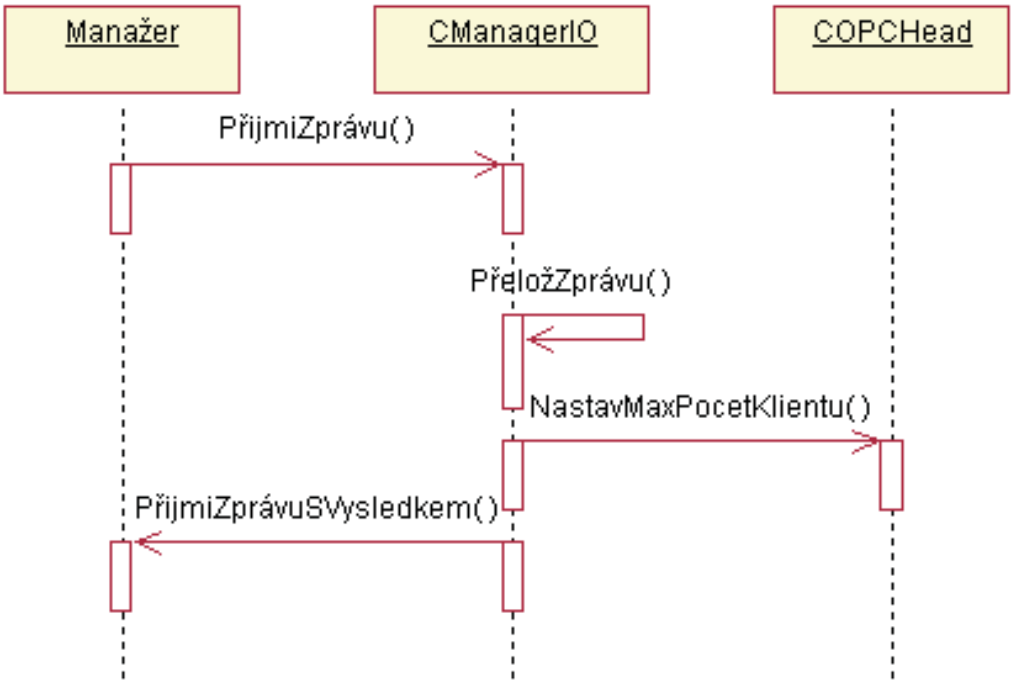
COPCServer je třídou zapouzdřující potřebná rozhraní komponenty OPC Server, popsaná v podkap. 4.3.1. Přehled funkcí třídy dává tab. 6.2. Příklad sekvence operací při založení skupiny je na obr. 6.4.

Počet skupin, které může klient vytvořit v jedné instanci třídy COPCServer je omezen na 10.

6.1.2.5 COPCGroup (*OPCGroup.h*)

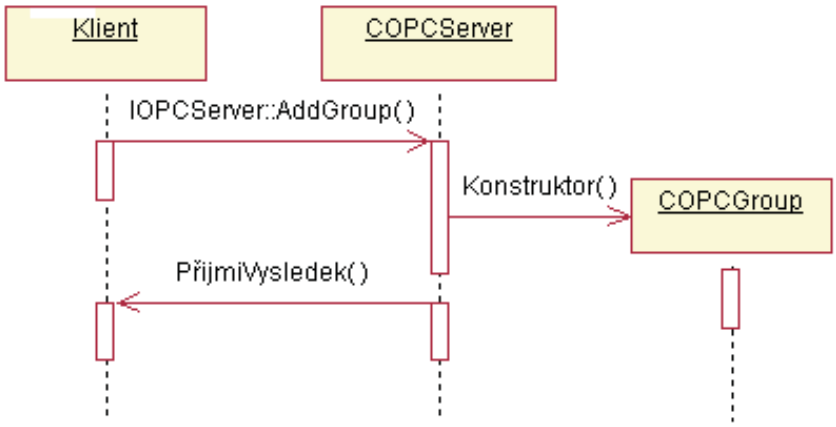
Tato třída implementuje všechna potřebná rozhraní OPC skupiny, viz podkap. 4.3.2. Metody shrnuje tab. 6.3.

Příklad 6.2 (Zpracování požadavku na počet max. klientů):



Obrázek 6.3: Zpracování požadavku na počet max. klientů

Příklad 6.3 (Založení skupiny):



Obrázek 6.4: Založení skupiny

Účel	Umístění
Metody rozhraní IUnknown	OPCServer.cpp
M. r. IOPCServer	OPCServer.cpp
M. r. IConnectionPointContainer	ConnectionPointContainer.cpp
M. r. IConnectionPoint	ConnectionPoint.cpp
M. r. IOPCCommon	OPCCommon.cpp
M. r. IOPCItemProperties	OPCItemProperties.cpp
M. r. IOPCBrowseAddressSpace	OPCBrowseAddressSpace.cpp
Další podpůrné metody	OPCServer.cpp

Tabulka 6.2: Skupiny metod třídy COPCServer

Účel	Umístění
M. r. IUnknown	OPCGroup.cpp
M. r. IOPCGroupStateMgt	OPCGroupStateMgt.cpp
M. r. IConnectionPointContainer	ConnectionPointContainer.cpp
M. r. IConnectionPoint	ConnectionPoint.cpp
M. r. IOPCSyncIO	OPCSyncIO.cpp
M. r. IOPCASyncIO2	OPCASyncIO2.cpp
M. r. IOPCItemMgt	IOPCItemMgt.cpp
Další podpůrné metody	OPCGroup.cpp

Tabulka 6.3: Skupiny metod třídy COPCGroup

Každá skupina může obsahovat max. 200 OPC položek. Z toho může být max. 100 určeno pro zápis na zařízení a 100 pro čtení ze zařízení. Pro rychlejší vyhledávání jsou položky rozděleny do dvou polí podle typu operace (čtení/zápis).

Asynchronní požadavky klientů jsou vyřízeny takto

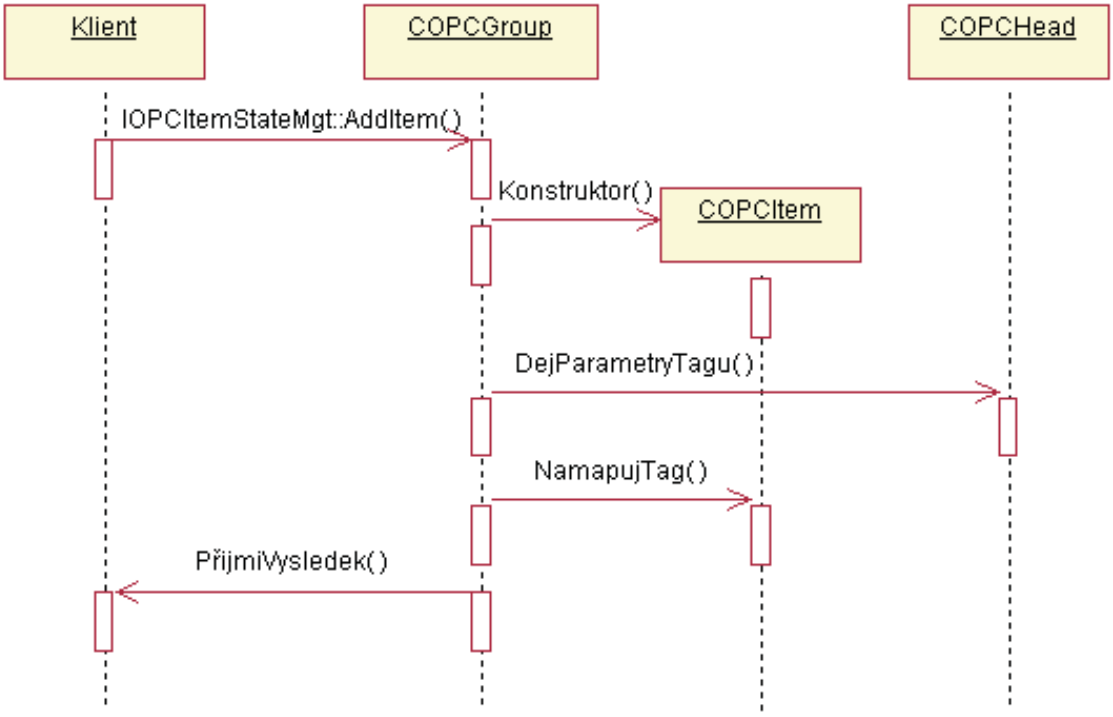
1. Klient zavolá některou z metod rozhraní IOPCAsyncIO2 (Read, Write, Refresh2).
2. Server uloží data od klienta do speciální třídy CTransactionData. Ta obsahuje základní data o položkách, se kterými chce klient pracovat.
3. Server uloží instanci CTransactionData do fronty požadavků a signalizuje událost, že se tak stalo.

4. Událost spustí uspané vlákno, které se stará o asynchronní požadavky. Toto vlákno je součástí třídy COPCHead. Vlákno spustí sekvenci operací které zapíší/přečtou data.

5. Vlákno použije metody rozhraní IOPCDataCallback k vrácení výsledků klientovi. Pokud fronta obsahuje další požadavky od jiných klientů, vlákno pokračuje v jejich zpracování, jinak samo přejde do uspaného stavu.

Dále jsem měl v plánu optimalizovat automatickou zpětnou vazbu. To se mi ale pořádně nezdařilo. Optimalizovaná verze byla v testech nestabilní. Server tedy po aktualizaci tagů prochází sekvenčně všechny položky všech skupin a klientů a zjišťuje, zda změnily hodnotu, pak pošle výsledky klientovi. Optimalizovaná verze severu si vytvářela speciální seznamy změněných položek. Vlákno, které se stará o automatickou aktualizaci, pak procházelo pouze tyto změněné položky. Příklad sekvence operací při založení položky je na obr. 6.5.

Příklad 6.4 (Založení položky):



Obrázek 6.5: Založení položky

6.1.2.6 COPCItem (*OPCItem.h*)

COPCItem zastupuje třídu pro OPC položku, viz podkap. 4.3.3. Kromě datových položek obsahuje taky některé funkce pracující s těmito položkami.

6.1.2.7 CManagerIO (*ManagerIO.h*)

Třída CManagerIO zprostředkovává komunikaci s manažerem serveru. Přijímá zprávy z pojmenované roury, dekóduje je dle komunikačního protokolu (viz podkap. 6.2.1.9) a volá potřebné metody třídy COPCHead. Na každý požadavek manažera pak zašle odpověď s chybovým kódem, popř. dalšími daty. Třída automaticky založí pojmenovanou rouru a vlákno, které periodicky zjišťuje zda se připojil manažer, či zaslal-li nějaký požadavek. Příklad na obr. 6.3 ukazuje sekvenci operací při vyřízení požadavku na změnu zařízení.

6.1.2.8 CLog (*Log.h*)

CLog zajišťuje záznam chybových zpráv do textového souboru. Jméno souboru je pevně nastaveno na *Log.txt*. Formát dat je jednoduchý, každý chybový záznam je na jednom řádku. Struktura chybového záznamu je na obr. 6.6.

```
Datum : Čas : Klient #: Chybová hláška : Číslo chyby
```

Obrázek 6.6: Struktura chybového záznamu

V některých případech záznam neobsahuje číslo klienta. Je to tehdy, když se chyba vyskytne v některých globálních funkcích, které nemají vztah k připojenému klientovi (např. nepodařilo se inicializovat kartu). Velikost souboru pro záznamy lze měnit pomocí manažera.

6.2 Manažer serveru

Konzolová aplikace serveru jistě efektivně využívá systémových prostředků ale není příliš uživatelsky přívětivá. Proto jsem k ní vytvořil ještě konfigurační prostředí - manažera (*OPCManager.exe*). Manažer je vytvořen v jazyce C# na platformě .NET a je aplikací s grafickým uživatelským rozhraním. Umožňuje jednoduchou konfiguraci některých parametrů OPC serveru.

Ke komunikaci s OPC serverem přicházelo v úvahu samozřejmě více typů meziprocové komunikace. Jsou to RPC, COM, DDE, TCP/IP či pojmenované roury. Všechny jsou efektivní, umožňují obousměrnou komunikaci a komunikaci na vzdálených počítačích. Implementace je nejjednodušší pro TCP/IP a pojmenované roury. Jelikož je manažer primárně určen ke konfiguraci serveru na stejném počítači, vybral jsem si pojmenované roury, u kterých firma Microsoft uvádí vyšší rychlosti. Hypotetická možnost vzdálené komunikace je s pojmenovanou rourou také možná, i když zde by bylo lepší použít přímo TCP/IP.

V manažerovi jsem implementoval následující funkce, které umožňují ovládat OPC server takto

- spustit server.
- odpojit všechny klienty.
- ukončit server.
- založit/odebrat tag z adresního prostoru serveru.
- nastavit max. počet připojitelných klientů k serveru.
- nastavit min. časovou konstantu pro aktualizaci adresního prostoru.
- možnost přepínat mezi kartami Advantech kompatibilními se serverem, pokud jich počítač obsahuje více.
- možnost nastavit zpracování přerušení karty.

Dále bylo potřeba implementovat komunikaci se serverem pomocí pojmenovaných rour v prostředí .NET C#. Manažer se skládá z těchto částí

- *MainForm* - hlavní okno aplikace.
- *FormAddTag* - okno umožňující založit nový tag adresního prostoru.
- *FormInterrupt* - okno pro nastavení použití přerušení.
- *FormLogging* - okno, které umožňuje nastavit záznam chyb do souboru.
- *FormSelectDevice* - okno pro výběr některé z nainstalovaných kompatibilních zařízení. Toto zařízení pak bude použito jako zdroj dat pro tagy adresního prostoru.

- *FormMaxClients* - okno pro nastavení max. počtu připojitelných klientů k serveru.
- *FormMinUpdateRate* - okno pro nastavení min. doby pro aktualizaci adresního prostoru.
- *NamedPipeNative* - třída tvořící obálku pro použití pojmenovaných rour.
- *PipeIO* - metody této třídy překládají uživatelské požadavky do zpráv pojmenované roury a posílají je serveru.
- *RegistryInterface* - třída, která slouží k práci z Windows registry.
- *FileIO* - třída umožňující uložení konfigurace do souboru.

Všechny třídy s grafickým rozhraním (okna) jsou vyobrazena a popsána v příloze B.1.

6.2.1 Hlavní části aplikace

6.2.1.1 MainForm (*MainForm.cs*)

Hlavní okno aplikace. Uživatel pomocí něho volí dostupné funkce manažera. Hlavní okno podle potřeby otvírá ostatní pomocná okna, viz ovládání programu v příloze B.1. Z těchto pomocných oken pak získává informace od uživatele a volá funkce, které zasílají zprávy serveru. V případě negativní odpovědi serveru zobrazí chybovou hlášku v logovacím okně. Příklad sekvence operací je na obr. 6.7.

6.2.1.2 FormAddTag (*FormAddTag.cs*)

Umožňuje založit tag. Uživatel má možnost vybrat zda bude tag určen pro zápis či čtení ze zařízení a jestli budou jeho hodnoty simulovány. Při simulaci server místo čtení z reálného zařízení náhodně generuje čísla. Simulace hodnot je možná pouze pro tagy určené ke čtení.

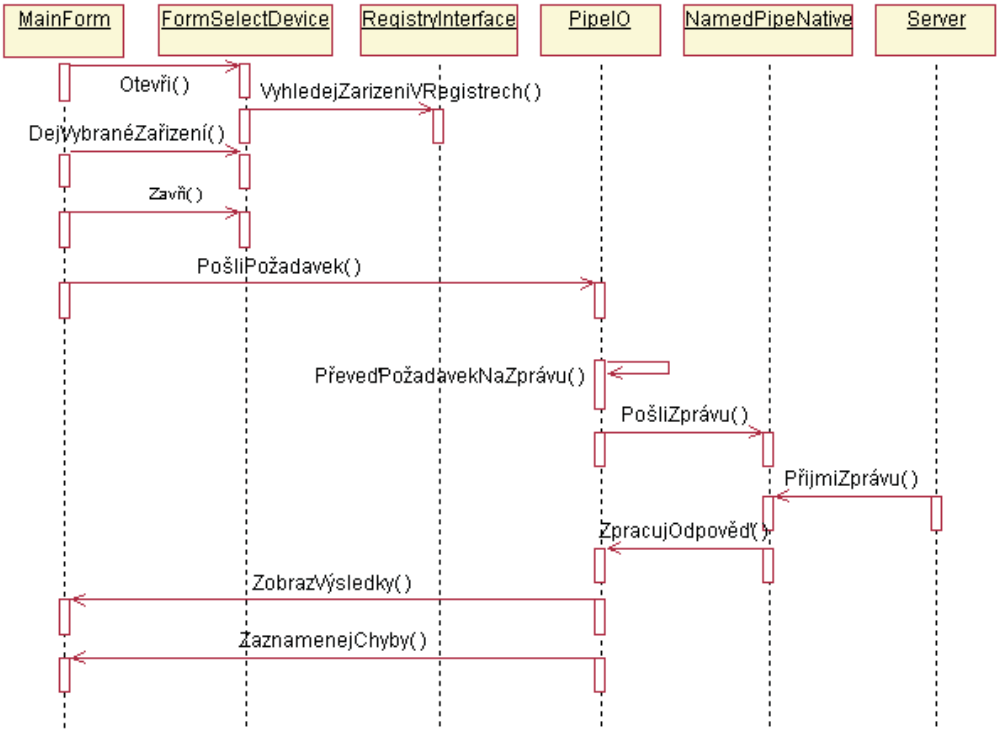
6.2.1.3 FormInterrupt (*FormInterrupt.cs*)

Okno které umožňuje uživateli povolit periodická přerušení karty. Uživatel může nastavit frekvenci přerušení pomocí dvou čítačů C0 a C1, viz příloha A.4.

6.2.1.4 FormLogging (*FormAddTag.cs*)

Umožňuje povolit záznam chyb do sobouru, s možností nastavit velikost souboru.

Příklad 6.5 (Výběr zařízení pro OPC server):



Obrázek 6.7: Výběr zařízení pro OPC server

6.2.1.5 FormSelectDevice (*FormSelectDevice.cs*)

Slouží k výběru zařízení ze seznamu nainstalovaných zařízení v systému. Server podporuje karty typu Advantech PCI 1750, nebo Advantech DEMO kartu, viz příloha A. Jestliže v systému není ani jedna z těchto karet nainstalovaná, má smysl založit pouze tagy pro čtení se simulovanými hodnotami.

6.2.1.6 FormMaxClients (*FormMaxClients.cs*)

Další jednoduché okno, které slouží k zadávání max. počtu připojitelných klientů.

6.2.1.7 FormMinUpdateRate (*FormMinUpdateRate.cs*)

Okno k zadávání min. obnovovací doby (tj. max. obnovovací frekvence) adresního prostoru.

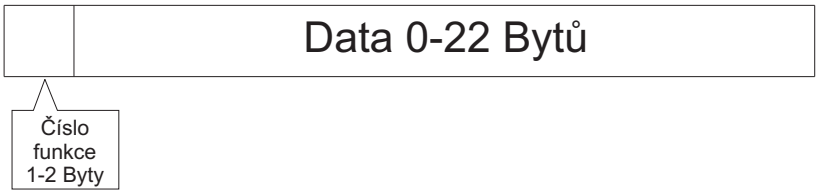
6.2.1.8 NamedPipeNative (*NamedPipeNative.cs*)

Tato třída tvoří obálku (viz podkap. 5.1) pro použití pojmenovaných rour. Obálka je nutná, protože platforma .NET Framework 1.1, kterou využívá jazyk C#, v sobě neimplementuje podporu pojmenovaných rour. K tomuto účelu jsem lehce upravil již hotovou obálku od autora Ivana Latunova, viz lit. [22]. Tuto obálku jsem vylepšil o podporu neblokovacího režimu pojmenovaných rour.

6.2.1.9 PipeIO (*PipeIO.cs*)

Hlavním úkolem třídy PipeIO je automaticky zabezpečit spojení se serverem. Zde dochází k překladu uživatelských požadavků do zpráv pojmenované roury dle komunikačního protokolu. Interně pak PipeIO využívá služeb obálky NamedPipeNative. Pojmenovaná roura funguje v neblokovacím režimu, viz podkap. 2.4.1.3. O automatické připojení a kontrolu stavu roury se stará samostatné vlákno, které je též součástí třídy PipeIO. Synchronizace přístupu vlákna ke sdíleným objektům jsem řešil pomocí *mutexu*. Mutex je obdoba kritické sekce z podkap. 6.1.1.1.

Pro účely komunikace jsem navrhl jednoduchý komunikační protokol. Obr. 6.8 ukazuje strukturu každé zprávy symbolizující některý z požadavků uživatele.



Obrázek 6.8: Obecná struktura zprávy s požadavkem

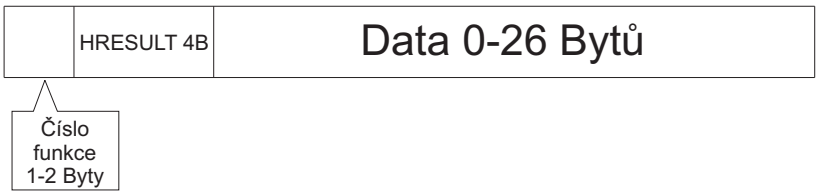
Možné uživatelské požadavky jsou uvedeny v tab. 6.4. K nim odpovídající struktury pole *Data* obsahuje tab. 6.5.

Na každý požadavek klienta server odpoví. Obrázek 6.9 ukazuje obecnou strukturu této zprávy

Číslo funkce v prvním nebo prvních dvou bytech se shoduje s číslem funkce požadavku. Chybový kód HRESULT, viz pokap. 3.3.1 může nabývat pro různé požadavky různých hodnot. Seznam všech možných kódů HRESULT pro každý požadavek je v tab. 6.7. Hodnoty kódů HRESULT a jejich význam je pak v tab. 6.8. Tabulka 6.6 ukazuje strukturu pole *Data* pro každou odpověď serveru.

Číslo funkce	Název
1	Přidej tag.
2	Ukonči server.
3	Zruš tag.
4	Vyber zařízení.
5	Nastav záznam chyb.
6	Odpoj klienty.
7	Nastav max. počet klientů.
8	Nastav max. frekvenci aktualizace adresního prostoru.
9	Nastav přerušení.
20 — 1	Aktualizuj info o tagu.
20 — 2	Aktualizuj info o zařízení.
20 — 3	Aktualizuj info o přerušení.
20 — 4	Aktualizuj info o záznamu chyb.
20 — 5	Aktualizuj info o max. počtu klientů.
20 — 6	Aktualizuj info o max. frekvenci aktualizace adresního prostoru.

Tabulka 6.4: Označení možných uživatelských požadavků



Obrázek 6.9: Obecná struktura zprávy s výsledkem

Po zjištění, že server založil pojmenovanou rouru, klient aktualizuje všechny hodnoty pomocí funkcí s prefixem 20. Další podobnou aktualizaci provádí klient po jakémkoli uživatelském požadavku.

6.2.1.10 RegistryInterface (*RegistryInterface.cs*)

Tato třída slouží k zápisu a čtení informací z registru Windows. Registr obsahuje nejen základní informace o OPC serveru, viz příloha D, ale také základní informace

Číslo funkce	Data	Celkem bytů
1	jméno t. 20B — čtení/zápis 1B — simulace 1B	23
2	N/A	1
3	číslo tagu 4B	5
4	číslo zařízení 4B	5
5	povol/zakaž 1B — velikost souboru 4B	6
6	normálně/bezpodmínečně 1B	2
7	počet klientů 4B	5
8	časová konstanta 4B	5
9	povol/zakaž 1B — freq0 4B — freq1 4B	10
20 — 1	index tagu 1B	3
20 — 2	N/A	2
20 — 3	N/A	2
20 — 4	N/A	2
20 — 5	N/A	2
20 — 6	N/A	2

Tabulka 6.5: Struktura datových polí pro uživatelské požadavky

o nainstalovaných zařízeních Advantech a jejich parametrech, viz příloha A.7.

6.2.1.11 FileIO (*FileIO.cs*)

Třída FileIO zajišťuje uložení konfigurace do souboru formátu XML. Jméno souboru je pevně nastaveno na *Config.xml*. .NET Framework 1.1 má v sobě implementovanou podporu formátu XML, práce s ním je tedy v C# bezproblémová. Strukturu konfiguračního souboru jsem navrhl také velice jednoduchou a je na obr. 6.10.

6.3 OPC Klient

Pro vytvoření klienta (*OPCClient.exe*) jsem využil opět jazyk .NET C#. Abych si dále usnadnil práci, použil jsem knihovny poskytované organizací OPC Foundation. Tyto knihovny v sobě obsahují funkce tvořící dohromady programátorské rozhraní, které umož-

Číslo funkce	Data	Celkem bytů
1	číslo tagu 4B	9
2	N/A	4
3	N/A	4
4	N/A	4
5	N/A	4
6	N/A	4
7	N/A	4
8	N/A	4
9	N/A	4
20 — 1	jméno 20B — číslo 4B — čtení/zápis 1B — simulace 1B	32
20 — 2	číslo zařízení	10
20 — 3	povol/zakaž 1B — freq0 4B — freq1 4B	15
20 — 4	povol/zakaž 1B — velikost souboru 4B	11
20 — 5	max. počet klientů 4B	10
20 — 6	min. časová konstanta	10

Tabulka 6.6: Struktura datových polí pro odpovědi na uživatelské požadavky

ňuje jednoduchou implementaci klienta/serveru (viz OPC .NET API na obr. 4.3).

Klient je opět grafická uživatelská aplikace velmi podobná manažeru. Skládá se především z hlavního okna, které zobrazuje všechny důležité informace, jako jsou OPC skupiny a položky vytvořené v serveru, aktuální stav položek a tabulku pro záznam chyb, viz ovládání klienta v příloze B.2.

OPC klient vytvořený v této práci je velice jednoduchý a hodí se spíše jen k demonstracím komunikace pomocí OPC Data Access. Získaná data z OPC serveru pouze zobrazuje a nevyužívá je k dalším výpočtům, vizualizacím apod., jako tomu obyčejně bývá u OPC klientů v průmyslové automatizaci. Klient implementuje pouze tyto funkce

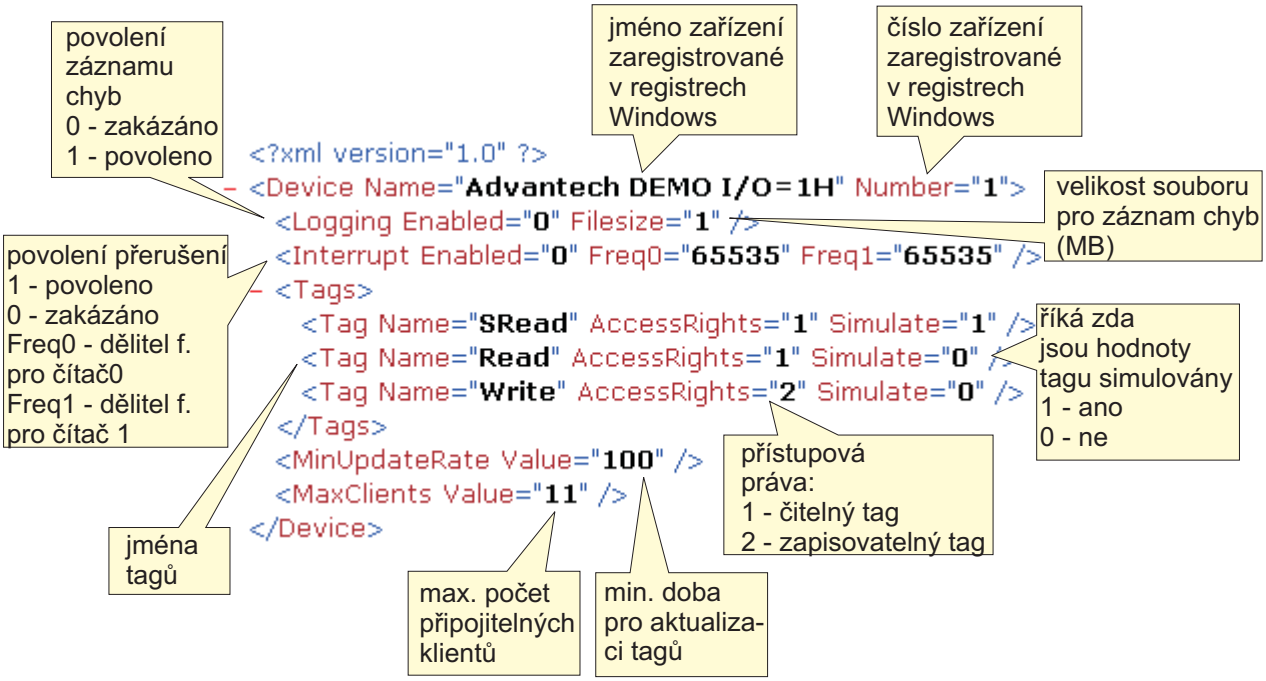
- připojení a zobrazení informací o OPC serveru.
- přidání OPC skupiny, uživatel může nastavit periodu čtení jejích OPC položek.
- synchronní čtení/zápis OPC položek. Čtení je provedeno s požadavkem současné aktualizace adresního prostoru.

Číslo funkce	Symbolický název	Číslo funkce	Symbolický název
1	S_OK E_OUTOFMEMORY E_FAIL OPC_E_DUPLICATENAME	8	S_OK E_OUTOFRANGE E_TOOHIGH E_FAIL
2	S_OK E_CLIENTSCONNECTED	9	S_OK E_FAIL E_OUTOFRANGE REGDB_E_KEYMISSING REGDB_E_READREGDB REGDB_E_WRITEREGDB ERROR_ACCESS_DENIED E_DEVICENOTREADY
3	S_OK E_TAG_CONNECTED E_FAIL		
4	S_OK E_DEVICENOTREADY E_FAIL		
5	S_OK E_OUTOFRANGE E_FAIL	20—1	S_OK E_LISTEMPTY E_FAIL
6	S_OK E_CLIENTSCONNECTED E_FAIL	20—2	S_OK E_FAIL
		20—3	S_OK E_FAIL
7	S_OK E_OUTOFRANGE E_FAIL E_MORECONNECTED	20—4	S_OK E_FAIL
		20—5	S_OK E_FAIL
		20—6	S_OK E_FAIL

Tabulka 6.7: Možné kódy HRESULT pro odpovědi na uživatelské požadavky

Hlavní části klienta jsou

- *MainForm (MainForm.cs)* - hlavní okno klienta. Podle potřeby otvírá ostatní pomocná okna (viz další text) a sbírá z nich data. Pro periodické čtení OPC položek nevytváří žádné zvláštní vlákno, ale využívá časovače Windows. Ten posílá hlav-



Obrázek 6.10: Struktura konfiguračního souboru

nímu oknu zprávy v daném intervalu. Na tyto zprávy hlavní okno reaguje čtením položek. Příklad sekvence operací je na obr. 6.11.

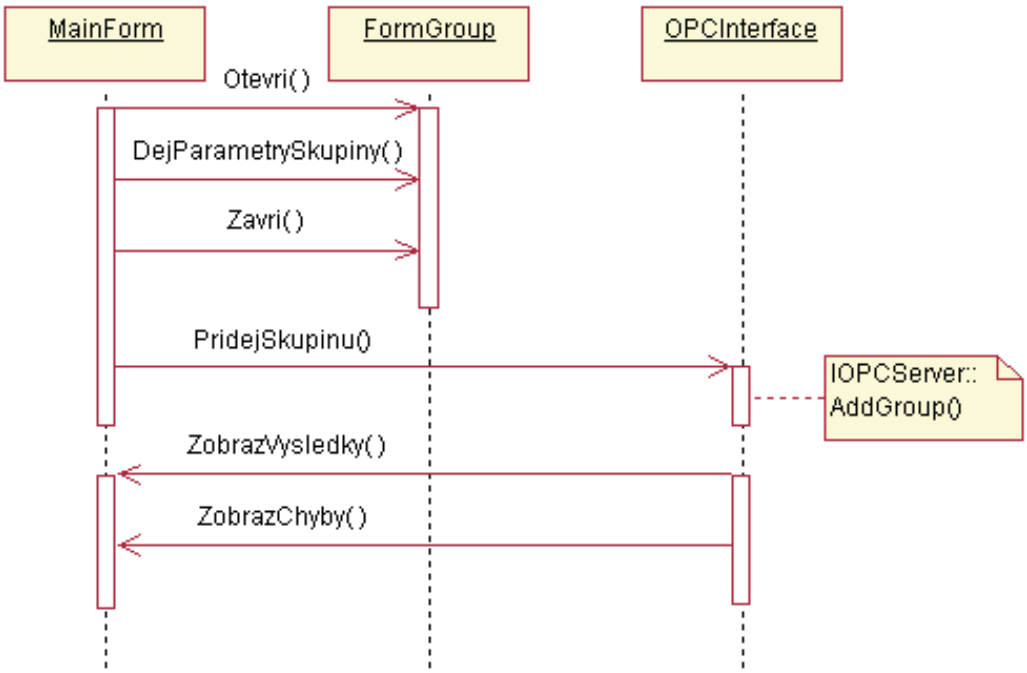
- *FormSelectServer (FormSelectServer.cs)* - okno, ve kterém uživatel zadá identifikátor serveru. Tento identifikátor, tzv. *ProgID*, lze pro každý korektně zaregistrovaný server nalézt ve registrech Windows. Záznam v registru pro můj server je v příloze D. Jelikož je klient primárně určen pro komunikaci se mým severem, je hodnota ProgID přednastavena na hodnotu pro můj server. Klienta lze samozřejmě také využít pro připojení k jiným serverům, stačí znát pouze ProgID každého z nich.

- *FormStatus (FormStatus.cs)* - toto okno slouží k zobrazení základních informací o připojeném serveru.

- *FormGroup (FormGroup.cs)* - okno k získání informací pro založení OPC skupiny. Jsou vyžadovány pouze dva parametry, a to jméno a interval pro čtení položek ve skupině. Oba tyto parametry lze měnit i po založení skupiny. Jméno skupiny musí být unikátní v rámci serveru. Toto pole lze ponechat prázdné, v tom případě OPC server vygeneruje jméno skupiny sám.

- *FormAddItem (FormAddItem.cs)* - okno k založení OPC položky. Jméno OPC položky se musí shodovat se jménem tagu na který se pak položka namapuje. Jména všech tagů adresního prostoru lze pro můj server zobrazit pomocí manažera.
- *FormWriteItem (FormWriteItem.cs)* - v tomto okně lze zadat libovolnou hodnotu. Tuto hodnotu pak klient zkusí zapsat do dané OPC položky. Jestliže server zjistí, že typ dat se neshoduje, pošle chybový kód. Klient tuto chybu zobrazí v tabulce chyb.
- *OPCInterface (OPCInterface.cs)* - třída která přijímá klientské požadavky od hlavního okna a převádí je na parametry metod OPC .NET API. Knihovny .NET API pak již zabezpečují meziprocesní komunikaci objektů COM přes RCW obálky, viz kap. 5.2.

Příklad 6.6 (Založení nové skupiny):



Obrázek 6.11: Založení skupiny

HRESULT	Hodnota	Význam
S_OK	0x00000000	Operace proběhla pořádku
E_OUTOFMEMORY	0x8007000	Nedostatek paměti pro provedení operace.
E_FAIL	0x8007000	Operace se nezdařila a důvod je blíže nespecifikován.
REGDB_E_READREGDB	0x80040150	Nelze přečíst registry Windows.
REGDB_E_WRITEREGDB	0x80040151	Nelze zapsat do registru Windows.
REGDB_E_KEYMISSING	0x80040152	Klíč v registru nenalezen.
ERROR_ACCESS_DENIED	0x00000005	Přístup do registrů odmítnut.
E_DEVICENOTREADY	0xC004040D	Zařízení není připraveno.
E_TAG_CONNECTED	0xC004040B	Tag nelze odstranit, je k němu připojen klient.
E_OUTOFRANGE	0xC004040C	Hodnota mimo povolené meze.
E_LISTEMPTY	0xC004040E	Požadovaný tag neexistuje.
E_CLIENTSCONNECTED	0xC0040411	K serveru jsou stále připojeni klienti.
E_MORECONNECTED	0xC0040411	K serveru je připojeno více klientů. Hodnota max. počtu klientů nemůže být nastavena na tuto hodnotu.
E_TOOHIGH	0xC0040412	Někteří již připojení klienti vyžadují vyšší frekvenci aktualizace adresního prostoru.
OPC_E_DUPLICATENAME	0xC004000C	Tag s tímto jménem již v adresním prostoru existuje.

Tabulka 6.8: Význam a hodnoty kódů HRESULT

Kapitola 7

Závěr

V teto práci se mi povedlo implementovat tři aplikace. Hlavní část práce, OPC server, jsem navrhl jako konzolovou aplikaci v jazyce C++. Server využívá základních funkcí Windows API a pro načtení aktuální konfigurace formát XML. Pro práci s formátem XML jsem použil již hotový software, viz lit. [21]. Server plně podporuje povinné části specifikace OPC Data Access 2.05, což jsem ověřil testem kompatibility navrženým organizací OPC Foundation. Z nepovinných částí specifikace jsem přidal rozhraní IOPCBrowseAddressSpace. Toto rozhraní je sice nepovinné, ale celá řada klientů jej de facto vyžaduje také. Z tohoto rozhraní jsem ale vypustil možnost filtrování tagů adresního prostoru serveru založené na regulárních výrazech. Nepodařilo se mi najít vhodné knihovny pro tento účel a vlastní implementace mi přišla nad rámec této práce. Periodická přerušení karty zmiňovaná v zadání práce server využívá s určitými omezeními. Jedná se hlavně o možnost použití pouze interního zdroje přerušení karty. Tyto omezení jsem zvolil po konzultaci s vedoucím práce. Plné využití přerušení karty by byla zbytečná práce, jelikož se s využitím této možnosti na katedře řízení do budoucna zatím nepočítá.

Za velkou výhodu považuji možnost propojení s prostředím MATLAB, hojně využívaného při návrhu řídících systémů. MATLAB obsahuje vestavěného OPC klienta, se kterým jsem úspěšně vyzkoušel čtení i zápis na kartu Advantech. Všechny testy lze nalézt v příloze C.

Na serveru je samozřejmě stále co vylepšovat. Jednalo by se hlavně o přidání dalších nepovinných částí specifikace OPC Data Access. Přes všechnu snahu se mi také nepodařilo dostatečně optimalizovat způsob zajištění automatické zpětné vazby informující klienta o změnách dat. Optimalizovaná verze serveru byla při testech značně nestabilní. Tento nedostatek se může projevit v případě, že server obsahuje mnoho čtecích datových položek, z nichž pouze některé změnily svoji hodnotu. U tohoto OPC serveru jsou zatím

všechny položky napojeny na stejných 16 vstupů karty, což znamená, že při jakékoliv změně na vstupu všechny položky změní svoji hodnotu stejně. Absence optimalizace by se tedy neměla projevit.

K demonstrační účelům jsem vytvořil jednoduchého klienta. Klient je grafická uživatelská aplikace implementovaná v jazyce C#. Práci jsem si ještě zjednodušil využitím .NET API poskytovaného organizací OPC Foundation. Tyto knihovny umožňují velmi snadnou implementaci klienta. Klient podporuje základní funkce OPC serveru, data umí číst a zapisovat pomocí synchronních operací. Primárně je tato aplikace určena pro použití se serverem z této práce, ale může samozřejmě spolupracovat i se servery jiných výrobců.

Aplikace manažera je poslední částí práce. Obsahuje grafické uživatelské rozhraní a je také implementovaná v jazyce C#. Umožňuje nastavit max. počet klientů připojitelných k serveru, konfigurovat adresní prostor, přepínat mezi zdroji zařízení (pokud např. počítač obsahuje více kompatibilních karet) a další globální nastavení serveru. Pro komunikaci se serverem jsem využil pojmenovaných rour. Navrhl sem také strukturu zpráv přenášených pojmenovanou rourou, tj. vlastní komunikační protokol. Komunikace funguje v asynchronním neblokovacím režimu pojmenované roury. Bylo nutno také implementovat .NET obálku pro pojmenované roury. Pro tento účel jsem upravil již hotovou obálku, viz lit. [22]. Manažera lze v budoucnu jednoduše rozšířit o více možných nastavení serveru.

Pro pohodlnou editaci kódu jsem použil vývojové prostředí Visual Studio .NET 2003.

Pro instalaci všech potřebných částí aplikací a podpůrných knihoven jsem vytvořil pomocí software Advanced Installer instalační program, viz lit. [23]. S jeho pomocí je nainstalování čí odebrání všech komponent jednoduchou záležitostí.

Všechny aplikace budou využity při výuce předmětů na katedře řízení FEL ČVUT.

Literatura

Tištěné monografie

[1] Kačmář, D. *Programujeme COM a COM+*. Praha: Vydavatelství ComputerPress, 2000. ISBN 80-7226-381-1.

[2] Robinson, S. et al.*C# - Programujeme profesionálně*. Praha:Vydavatelství ComputerPress, 2003. ISBN 80-251-0085-5.

[3] Virius, M. *Programování v C++*. Praha:Vydavatelství ČVUT, 2001. ISBN 80-01-01874-1

Elektronické monografie, články a odkazy

[4] Svoboda, T. *Průmyslová automatizace s využitím OPC*. Diplomová práce. Praha, 2003.

[5] Mastný, R. *Technologie COM a OPC*. Diplomová práce. Praha, 2002.

[6] *OPC Data Access Custom Interface Specification 2.05* [online]. OPC Foundation, 2002. Dostupný z WWW: www.opcfoundation.org

[7] Brockschmidt, K. *What is OLE really about* [online]. Microsoft Corporation, 1996. MSDN Library. Dostupný na WWW: www.msdn.com

[8] *TCP/IP Reference Page* [online]. Dostupný na WWW: www.protocols.com

[9] *Windows Sockets* [online]. Dostupný na WWW: www.sockets.com

[10] *MIDL Reference* [online]. Microsoft Corporation, 2006. MSDN Library. Dostupný na WWW: www.msdn.com

[11] *Connection Points* [online]. Microsoft Corporation, 2006. MSDN Library. Dostupný na WWW: www.msdn.com

[12] *COM Functions Reference* [online]. Microsoft Corporation, 2006. MSDN Library. Dostupný na WWW: www.msdn.com

[13] *Interprocess Communications* [online]. Microsoft Corporation, 2006. MSDN Library. Dostupný na WWW: www.msdn.com

[14] *Dynamic Data Exchange* [online]. RHA Minisystems Ltd. Dostupný na WWW: www.angelfire.com

[15] *IEnumUnknown* [online]. Microsoft Corporation, 2006. MSDN Library. Dostupný na WWW: www.msdn.com

[16] *IEnumString* [online]. Microsoft Corporation, 2006. MSDN Library. Dostupný na WWW: www.msdn.com

[17] *VARIANT and VARIANTARG* [online]. Microsoft Corporation, 2006. MSDN Library. Dostupný na WWW: www.msdn.com

[18] *Synchronization Reference* [online]. Microsoft Corporation, 2006. MSDN Library. Dostupný na WWW: www.msdn.com

Použitý Software

[19] *Visual Studio .NET 2003*. Microsoft Corporation, c2003-2006.

[20] *MATLAB*. The MathWorks Inc., c1984-2006

[21] Berghen, F. V. *Free C++ XML parser* [online]. Kranf Site. Dostupný na WWW: http://iridia.ulb.ac.be

[22] Latunov, I. *Inter-Process Communication in .NET Using Named Pipes* [online]. CodeProject, c2004. Dostupný na WWW: www.codeproject.com

[23] *Advanced Installer 3.9* [online]. Caphyon Ltd., c2002-2006. Dostupný na WWW: www.advancedinstaller.com

[24] *Matrikon OPC Explorer* [online]. Matrikon Inc., c2006. Dostupný na WWW: www.matrikonopc.com

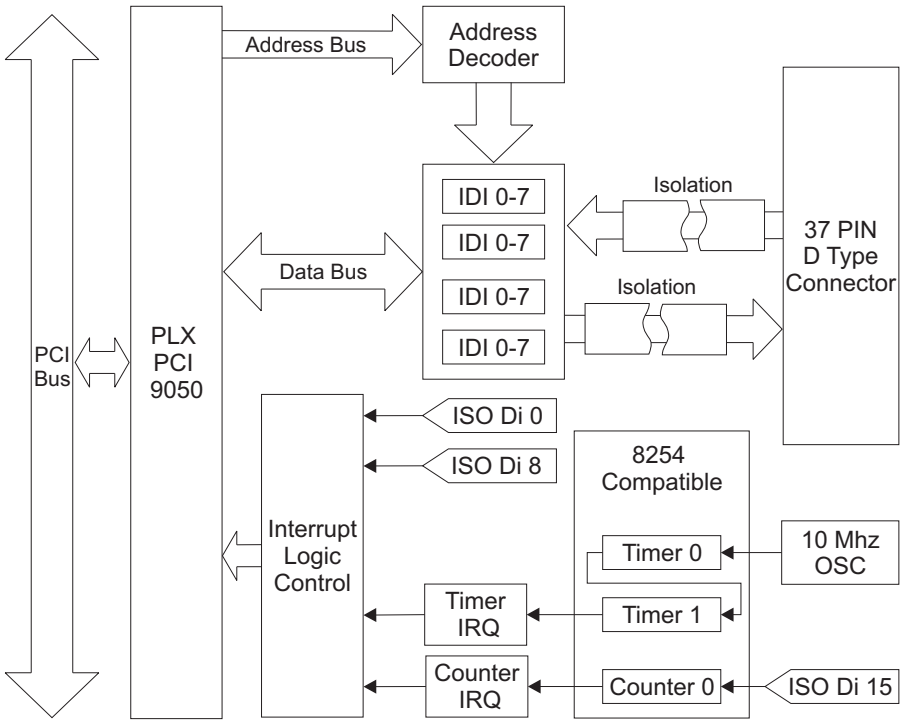
[25] *Visual OPCTestValidator* [online]. Terravic Corporation, c2001. Dostupný na WWW: www.opctest.com

[26] *LightOPC server* [online]. Lab43, c2000. Dostupný na WWW: www.ipi.ac.ru

Příloha A

Karta Advantech PCI 1750

OPC server vytvořený v této práci je určený k práci se vstupně-výstupní kartou firmy Advantech PCI 1750. Tato karta se hojně využívá v průmyslových aplikacích. Karta obsahuje 16 opticky izolovaných vstupů a stejný počet opticky izolovaných výstupů. Další uživatelsky zajímavou součásti je obvod Intel 8254 obsahující tři 16bitové konfigurovatelné čítače/časovače. Izolační ochrana karty je 2500 V_{DC}. Ke kartě je standardně dodáván Win32 DLL ovladač. Blokové schéma karty je na obr. A.1.



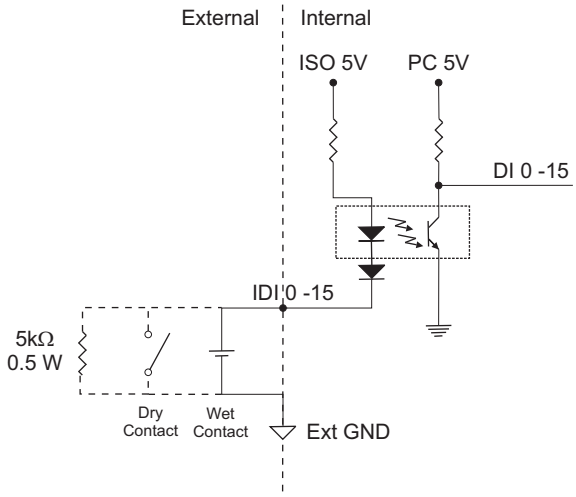
Obrázek A.1: Blokové schéma PCI 1750

A.1 Vstupy

Každý ze vstupů lze popsat schématem na obr. A.2.

Pokud prochází fotodiodou proud dostatečný k sepnutí fototranzistoru, na vstupu je úroveň 0. To lze zajistit dvěma způsoby

- suchý (*dry*) kontakt - uzemnění vstupu. Neuzemněný vstup dává úroveň 1.
- živý (*wet*) kontakt - připojení ext. zdroje napětí. Napětí zdroje 0-2V dává úroveň 0, napětí 5-48V dává úroveň 1. Jestliže je vnitřní odpor zdroje napětí větší než 5kΩ, může dojít k nesprávné funkci zařízení. K eliminaci takového odporu je vhodné připojit ke zdroji paralelně 5kΩ(0,5 W) rezistor.



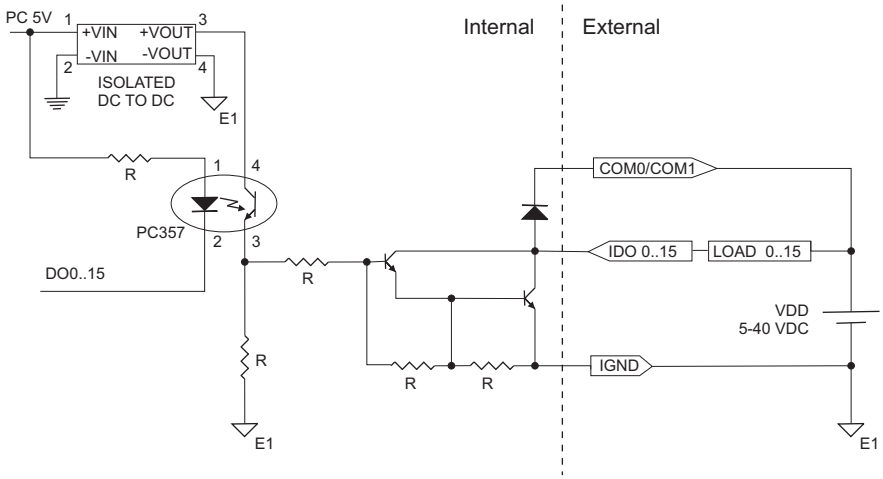
Obrázek A.2: Blokové schéma vstupu karty PCI 1750

A.2 Výstupy

Zapojení každého z výstupů je na obr. A.3. K sepnutí vnějších kontaktů je použita dvojice tranzistorů v Darlingtonově zapojení.

Výstupy 0-7 sdílí pin COM1, výstupy 8-15 sdílí pin COM2. Tyto piny je vhodné použít při připojení induktivní zátěže, k odvedení proudu způsobeného naindukovaným napětím. Indukované napětí zde vzniká při přepínání stavů výstupu.

Jestliže je k výstupům připojen zdroj napětí neměla by hodnota proudu tekoucího do karty překročit 200mA.



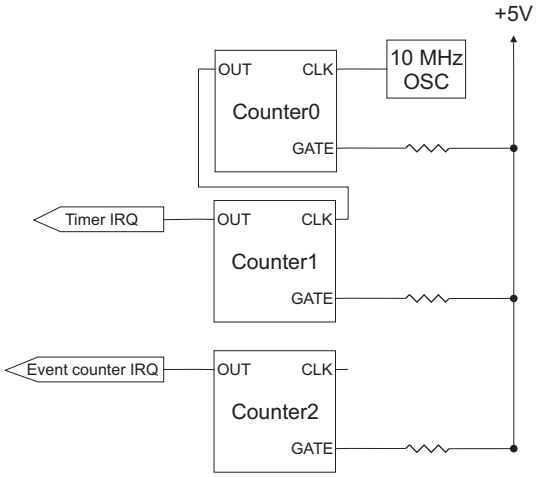
Obrázek A.3: Blokové schéma výstupu karty PCI 1750

A.3 Čítače, časovače

Schéma zapojení obvodu 8254 a jeho čítačů/časovačů je na obr. A.4. Čítače Counter0, Counter1 jsou zapojeny do kaskády a tvoří tak dohromady 32bitový čítač. Hodinovým vstupem pro Counter0 je vestavěný oscilátor s frekvencí 10Mhz. Praktický smysl využití těchto čítačů v kartě PCI 1750 je pouze ke generování periodického přerušení.

Dalším čítačem je Counter2. Jako hodinový vstup využívá vstup IDI15. Aplikace můžou tento čítač využít ke třem účelům

- generování periodického přerušení.
- počítání impulsů.
- měření frekvence impulsů.



Obrázek A.4: Blokové schéma zapojení čítačů karty PCI 1750

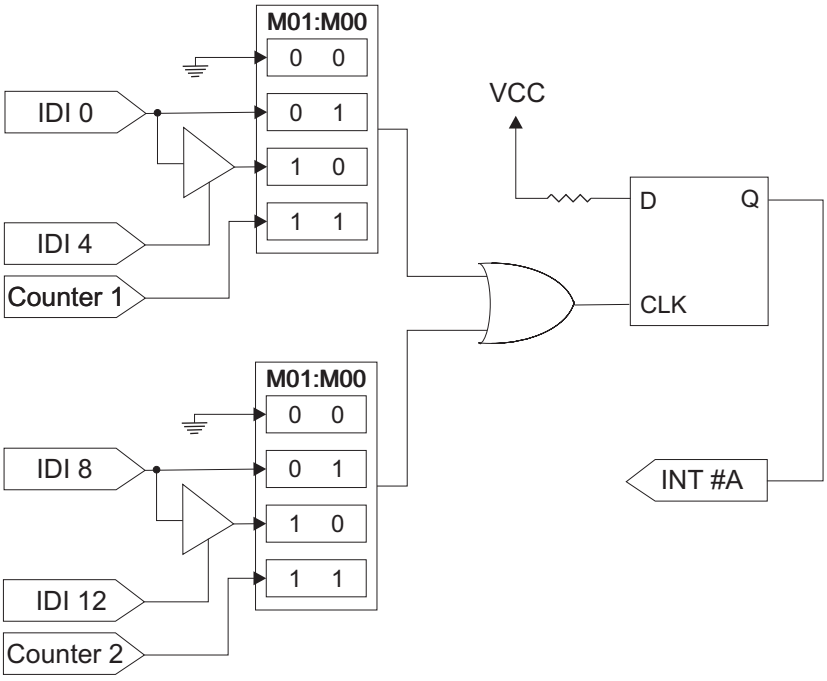
A.4 Přerušení

Karta PCI 1750 má od systému přiřazeno jedno hardwarové přerušení. Zdroj přerušení vybírá logika na obr. A.5.

Možných šest zdrojů přerušení je uspořádáno do dvou skupin. V každé skupině může být zdrojem přerušení max. jen jeden ze signálů, záleží na nastavení kontrolního registru skupiny. Prakticky tak výsledné přerušení generují dva z šesti signálů. Je-li zdrojem přerušení Counter2 či kaskáda Counter0+Counter1 může být ještě upraven dělitel externí frekvence každého čítače. Hodnota dělitele se pohybuje v rozmezí 2 až 65535. Dosažitelné frekvence pro každý zdroj přerušení zobrazuje tabulka A.1.

Zdroj	Vstupní f.	Dělitel	Výstupní f.
C0+C1	10Mhz	2^2 až 65535^2	0.002328Hz až 2.5Mhz
C2	0 až 1Mhz	2 až 65535	0 až 0.5Mhz
IDI0	0 až 10Khz	N/A	0 až 10Khz
IDI4	0 až 10Khz	N/A	0 až 10Khz
IDI8	0 až 10Khz	N/A	0 až 10Khz
IDI12	0 až 10Khz	N/A	0 až 10Khz

Tabulka A.1: Tabulka dosažitelných frekvencí z různých zdrojů přerušení



Obrázek A.5: Blokové schéma zapojení zdroje přerušení PCI 1750

A.5 Vybrané funkce DLL ovladače

OPC server vytvořený v této práci využívá Win32 DLL ovladač, standardně dodávaný firmou Advantech ke kartě. Ze všech dostupných funkcí karty využívá jen menší část. Tyto funkce jsou v tab. A.2.

A.6 Advantech DEMO karta

Ovladač firmy Advantech umožňuje nainstalovat v osobním počítači kromě fyzických zařízení také softwarovou kartu. Tato karta existuje pouze v operační paměti počítače. Nepodporuje sice všechny funkce ovladače ale měla pro mě praktické využití při testování OPC serveru. Jakákoliv hodnota zapsaná na na její výstup se okamžitě objeví na jejím vstupu. Lze tedy simulovat rychlé změny hodnot a vyzkoušet správnou funkci serveru

Název funkce	Popis
DRV_DeviceOpen	Inicializace karty.
DRV_DeviceClose	Uvolnění karty.
DRV_DeviceGetList	Vrací seznam nainstalovaných zařízení v systému, kompatibilních s DLL ovladačem.
DRV_DrvReadPortWord	Přečte hodnoty ze všech 16 vstupů karty.
DRV_WritePortWord	Zapíše hodnoty na všech 16 vstupů karty.
DRV_ReadPortByte	Přečte hodnoty 8 vstupů karty. Je potřeba vybrat správnou skupinu (karta má dvě).
DRV_WritePortByte	Zapíše hodnoty do vybrané skupiny 8 vstupů.
DRV_EnableEvent	Povolí/Zakáže příjem přerušení.
DRV_CheckEvent	Zjišťuje, zda přišlo přerušení v daném časovém intervalu.
DRV_GetErrorMessage	Převede číselný chybový kód na vysvětlující textový řetězec.

Tabulka A.2: Vybrané funkce DLL ovladače pro kartu Advantech PCI 1750

i v těchto extrémnějších podmínkách. Více o testování serveru je v příloze C.

A.7 Záznamy v registrech Windows o zařízeních Advantech

Po instalaci ovladačů pro karty Advantech se do registru Windows zapíší některé hodnoty parametrů těchto karet. Nejdůležitější záznamy v registrech jsou

- *HKEY_LOCAL_MACHINE \SYSTEM \CurrentControlSet \Services \ADSDAQ* - klíč obsahující všechny nainstalované zařízení Advantech. Podklíče pak obsahují

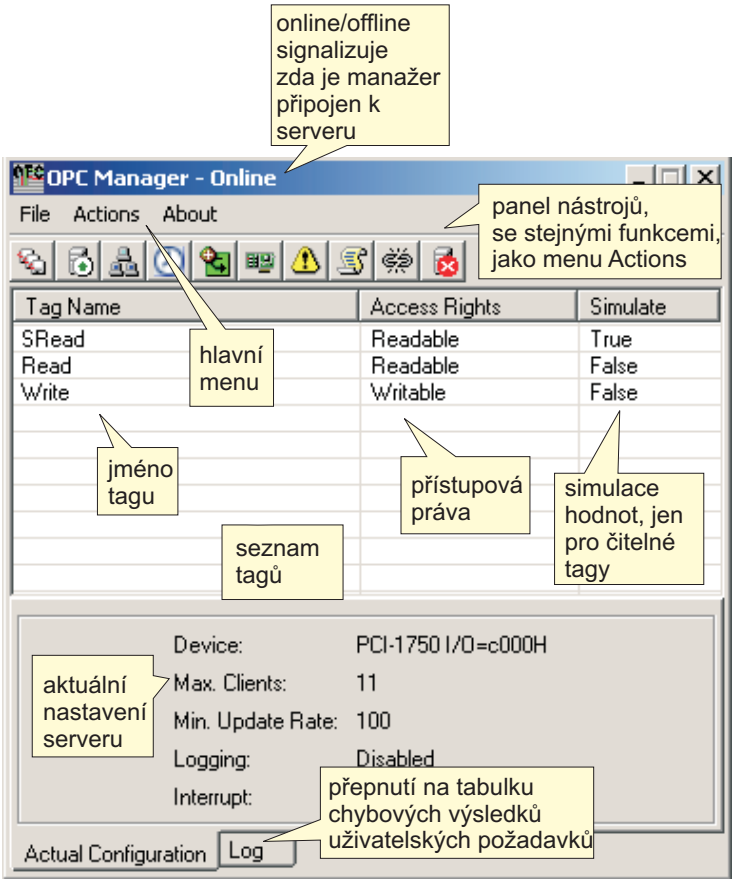
název každého zařízení spolu s názvem jeho ovladače.

- *HKEY_LOCAL_MACHINE \SYSTEM \CurrentControlSet \Services \ADS1750S \DeviceXXX* - kde XXX zastupuje číslo zařízení. Tento klíč se vztahuje k nainstalovanému zařízení typu Advantech PCI 1750. Jeho nejdůležitějšími hodnotami jsou
  - InterruptSource - nastavení zdroje přerušení.
  - Timer1Count - nastavení děliček frekvence vnitřního generátoru, tj. přednastavení čítačů C0 a C1.
  - Counter2Count - nastavení děličky frekvence vnějšího zdroje přerušení, tj. přednastavení čítače C2.

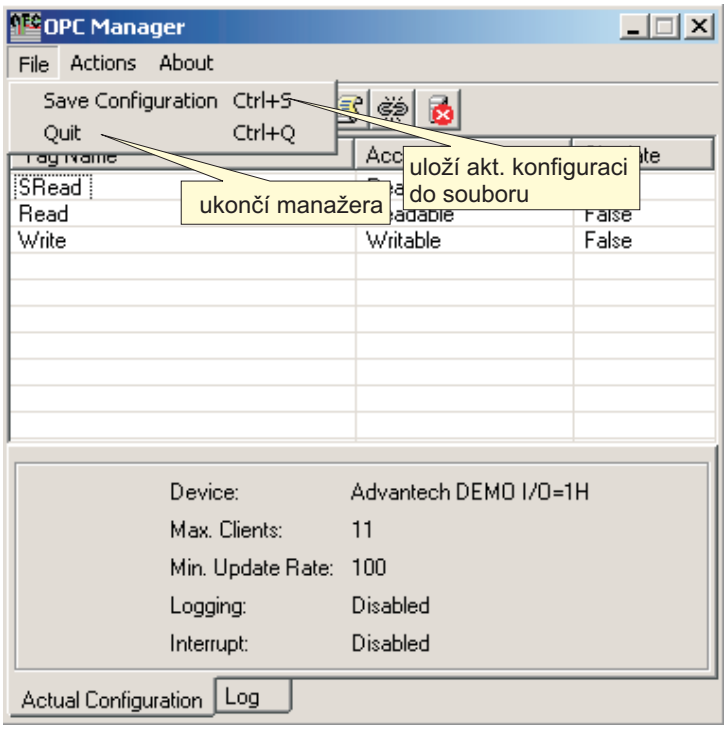
Příloha B

Ovládání aplikací

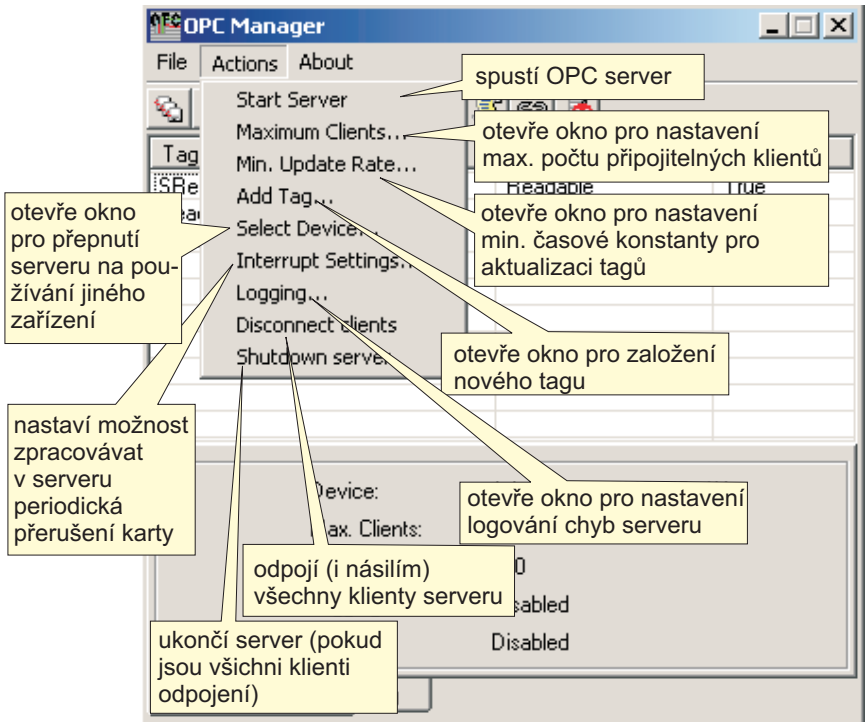
B.1 Manažer serveru



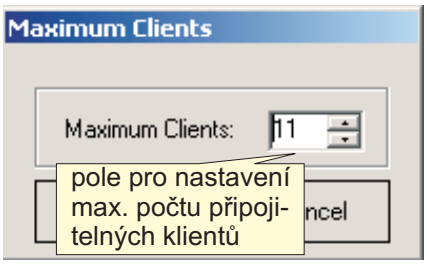
Obrázek B.1: Hlavní okno manažera



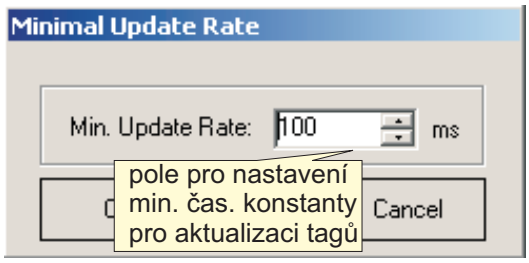
Obrázek B.2: Hlavní menu manažera - File



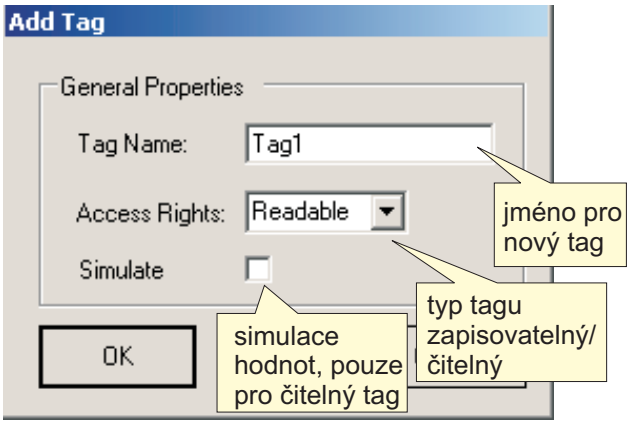
Obrázek B.3: Hlavní menu manažera - Actions



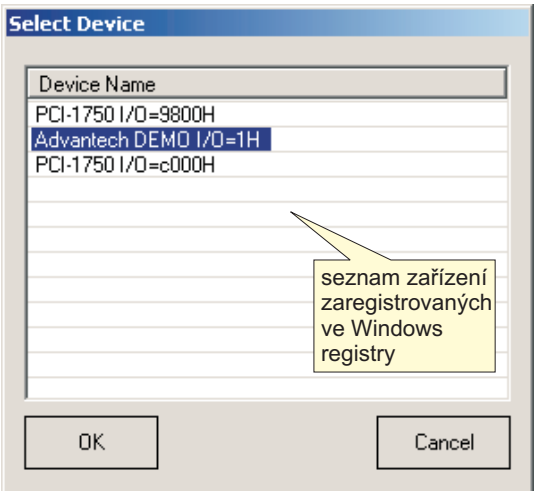
Obrázek B.4: Okno pro nastavení max. počtu klientů



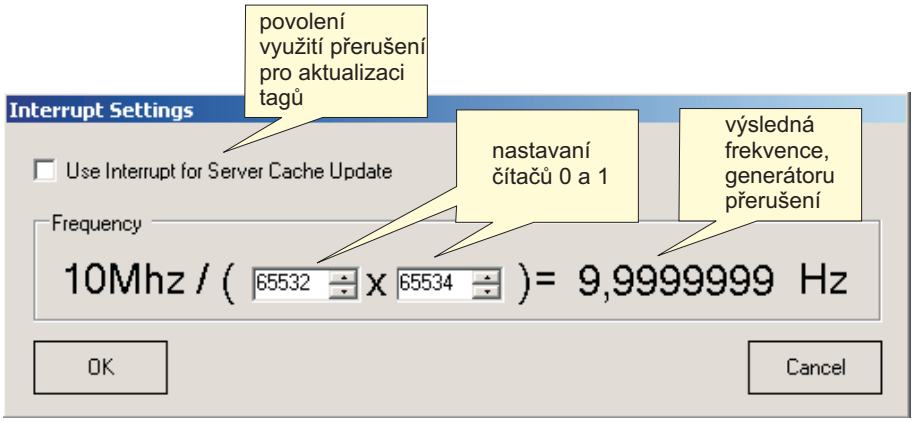
Obrázek B.5: Okno pro nastavení max. počtu klientů



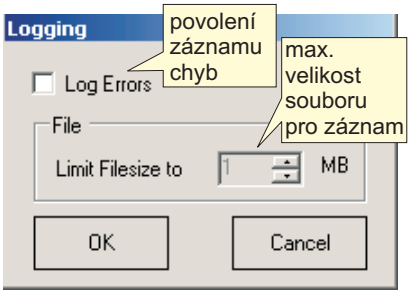
Obrázek B.6: Okno pro založení nového tagu



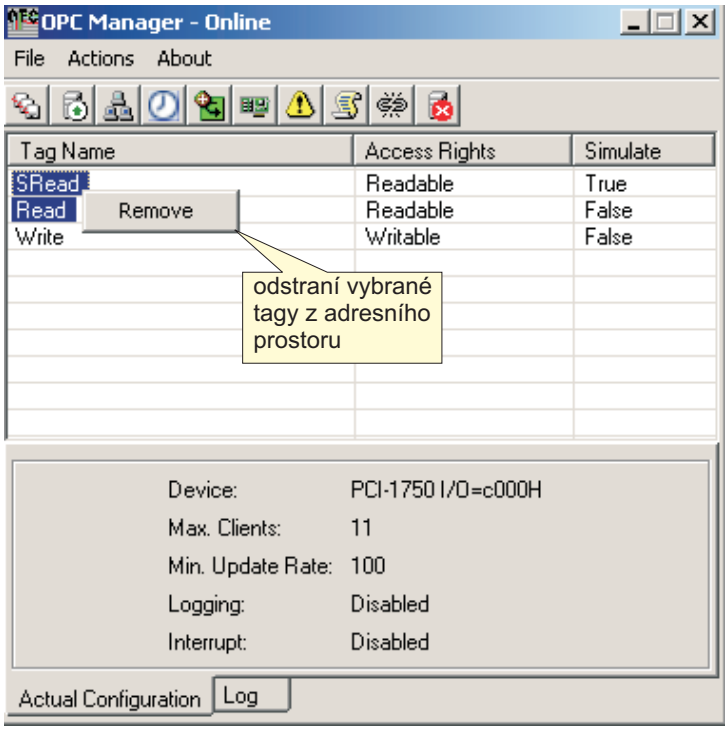
Obrázek B.7: Okno pro změnu zařízení pro OPC server



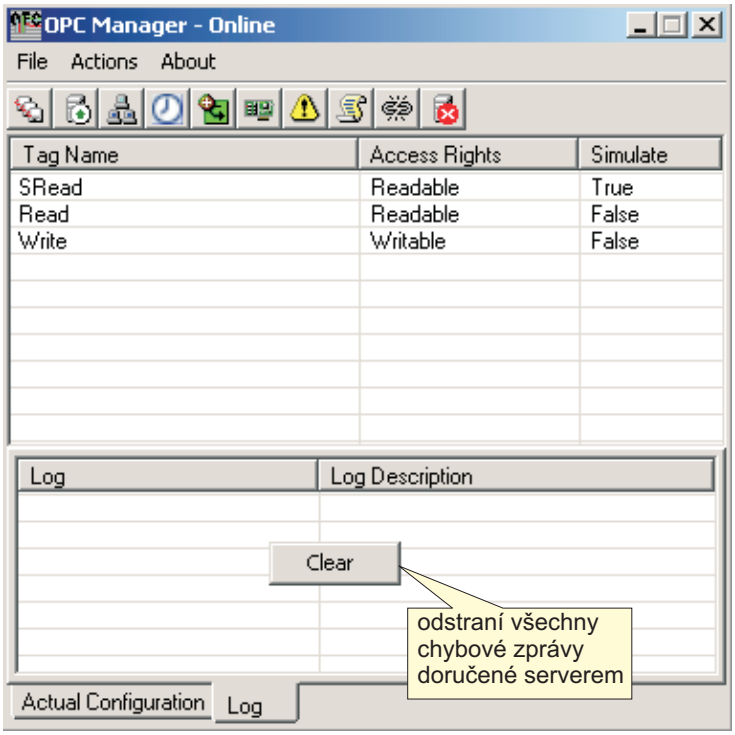
Obrázek B.8: Okno pro nastavení přerušení



Obrázek B.9: Okno pro nastavení záznamu chyb

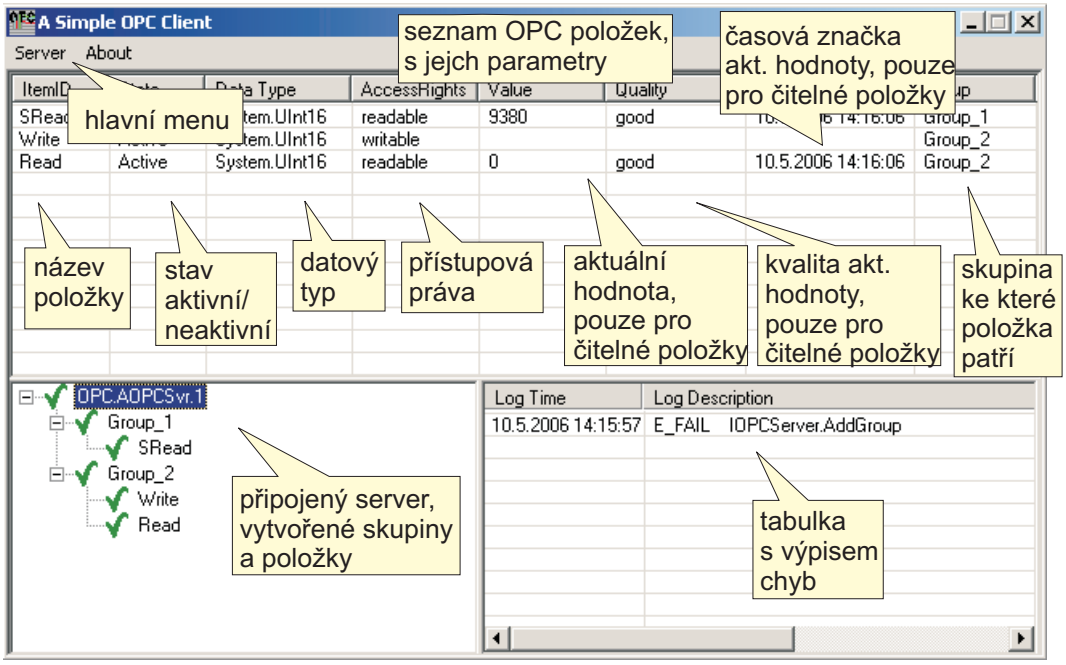


Obrázek B.10: Kontextové menu pro adresní prostor

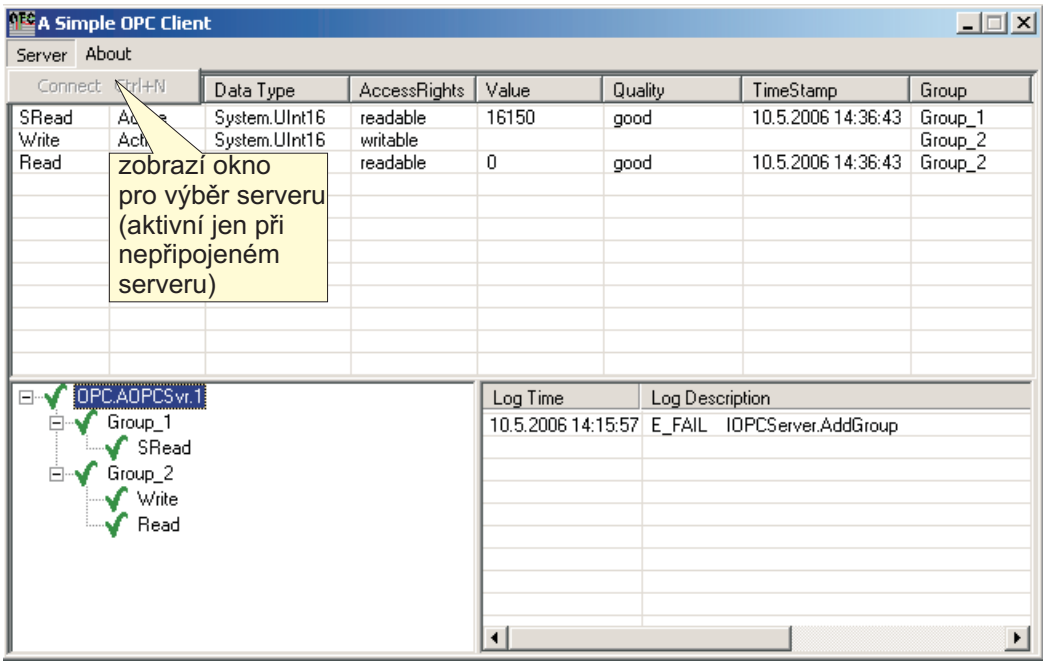


Obrázek B.11: Kontextové menu pro záznam chybových výsledků uživatelských požadavků

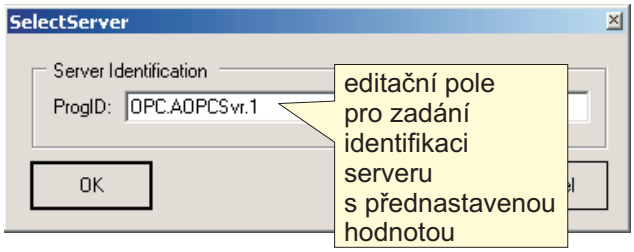
B.2 Klient



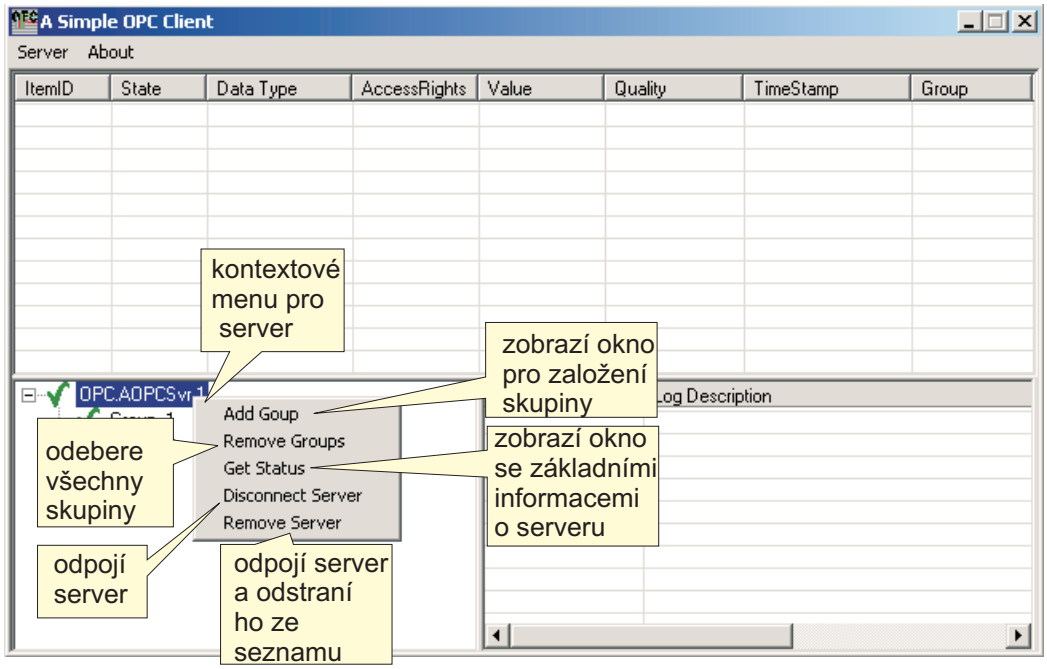
Obrázek B.12: Hlavní okno klienta



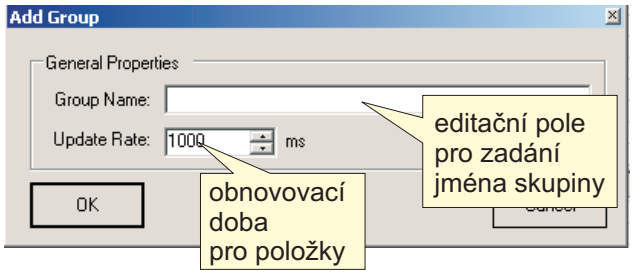
Obrázek B.13: Hlavní menu klienta



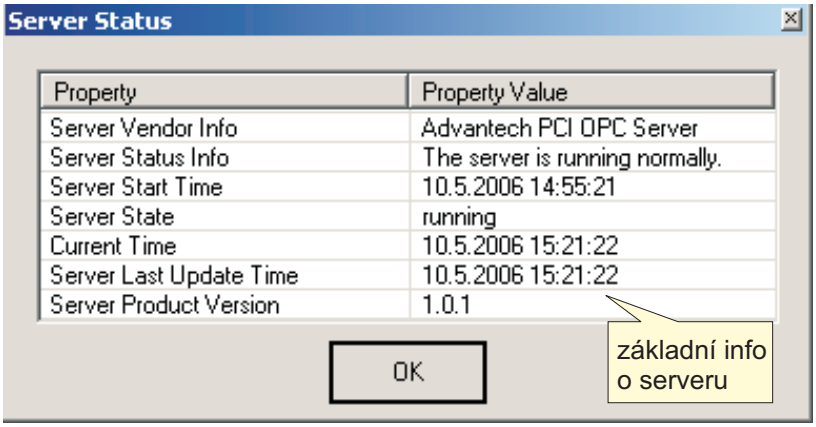
Obrázek B.14: Okno pro zadání ProgID serveru



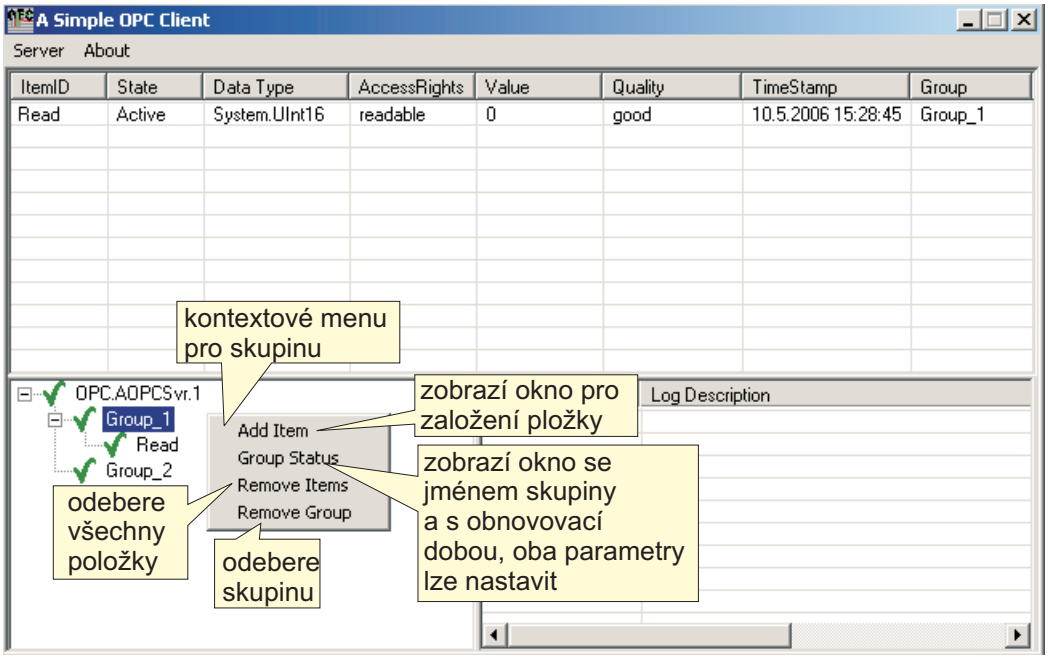
Obrázek B.15: Okno pro zadání ProgID serveru



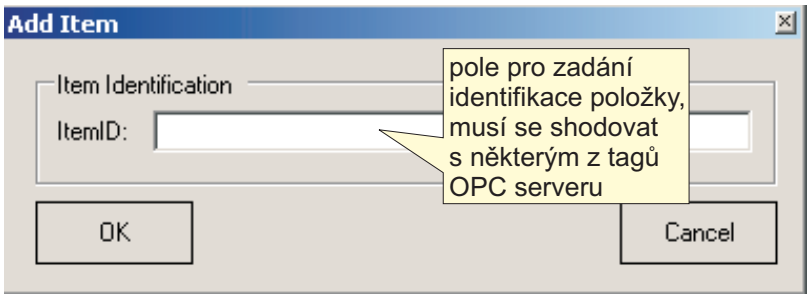
Obrázek B.16: Okno pro založení skupiny



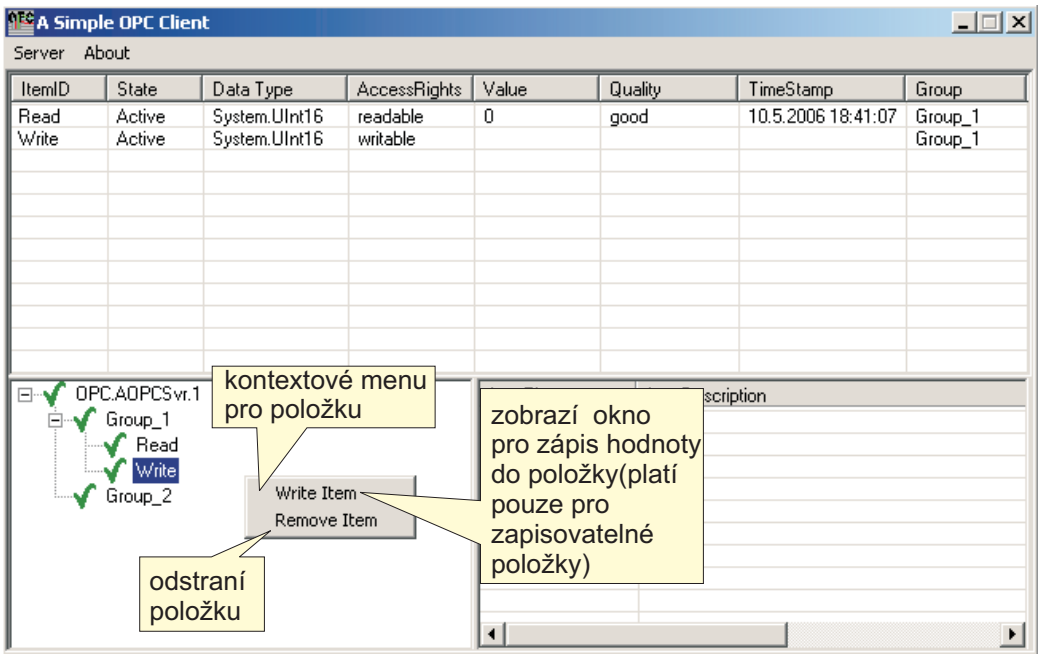
Obrázek B.17: Okno ze základními parametry serveru



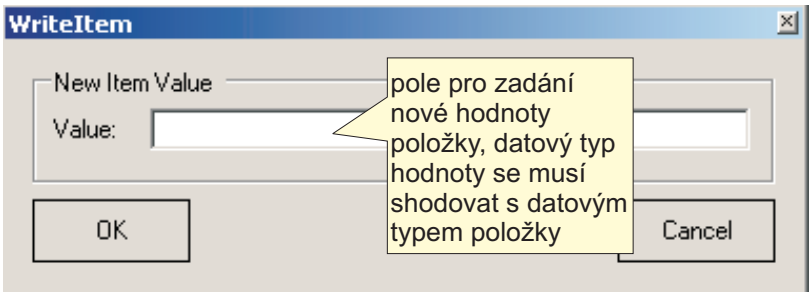
Obrázek B.18: Kontextové menu pro skupinu



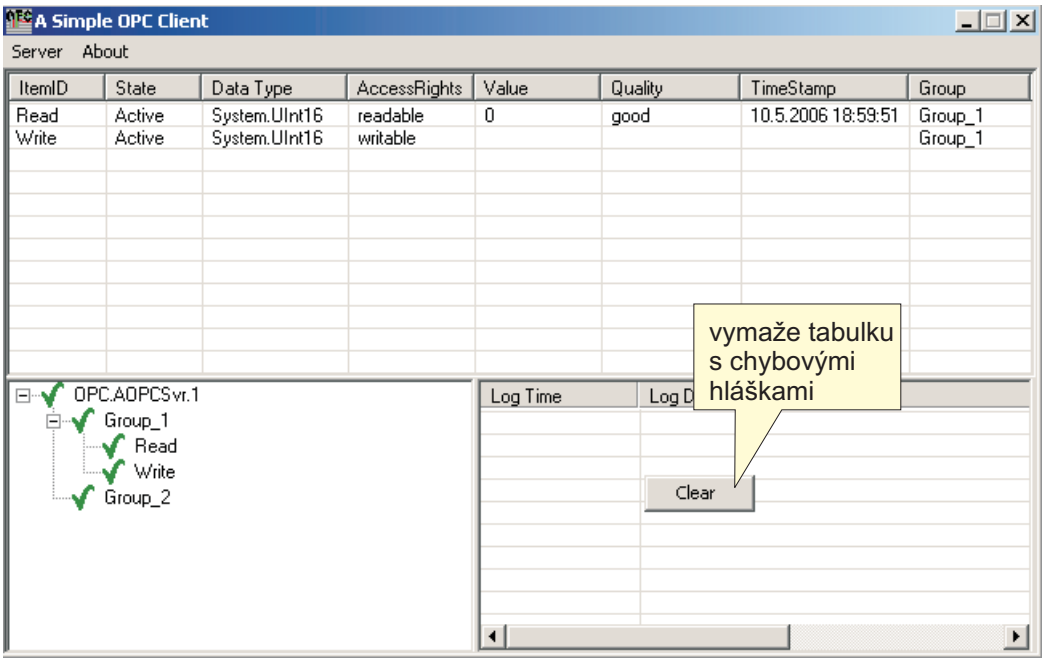
Obrázek B.19: Okno pro založení nové položky



Obrázek B.20: Kontextové menu pro položku



Obrázek B.21: Okno pro zadání nové hodnoty položky



Obrázek B.22: Kontextové menu pro tabulku chyb

B.3 Server

Server je konzolová aplikace a všechny její funkce jsou volány vzdáleně pomocí OPC klientů, či manažera serveru.

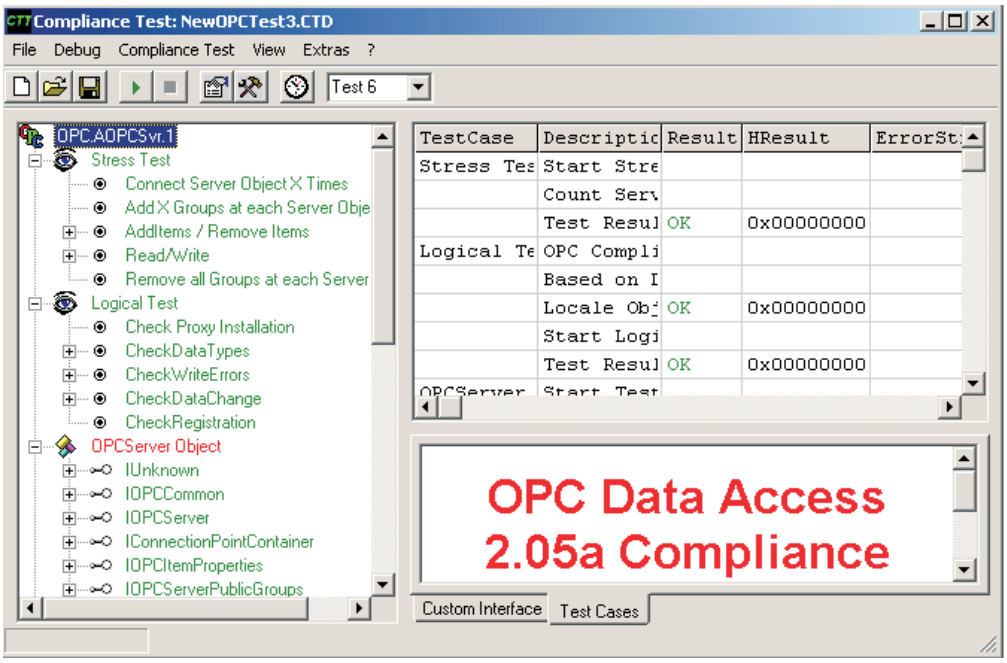
Příloha C

Testy aplikací

C.1 Server

C.1.1 OPC Compliance Test

Prvním testem bylo ověření kompatibility se specifikací OPC Data Access 2.05. K tomuto účelu jsem použil nástroj vyvinutý organizací OPC Foundation zvaný OPC Compliance Test Tool.



Obrázek C.1: Hlavní okno OPC Compliance testu

Hlavní části testu kompatibility

- *Stress Test* - rychlé připojení/odpojení 10 klientů, pro každého klienta přidáno/odebráno 10 skupin, pro každou skupinu přidáno/odebráno 10 položek.
- *Logical Test* - kontrola správné registrace serveru, základního zápisu/čtení, podporovaných datových typů, automatické zpětné vazby.
- *OPC Server Test* - test všech rozhraní komponenty OPC Server.
- *OPC Group Test* - test všech rozhraní komponenty OPC Group.
- *Performance Test* - měření rychlosti čtení/zápisu serveru.

Výsledky testu jsou uvedeny v tab. C.1

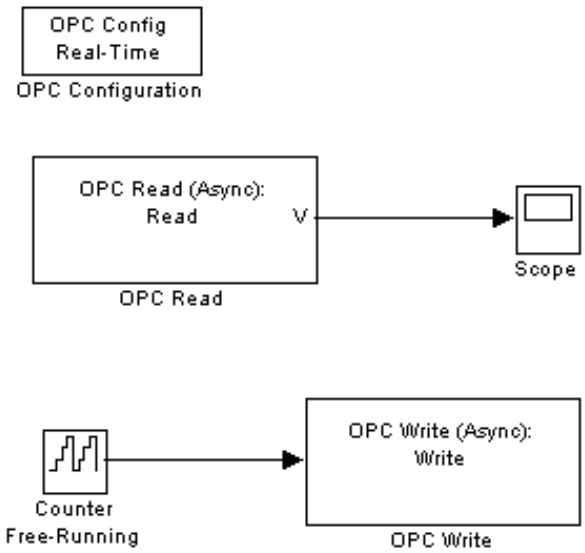
TestCase	Description	Result
Stress Test	Start Stress Test	
	Count Server = 10, Count Groups = 10, Count Items = 10	
	Test Result Stress Test	OK
Logical Test	OPC Compliance Test Program	
	Based on Data Access Custom Interface Specification 2.05	
	Locale Object (with IUnknown) Created for OPC.AOPCSvr.1	OK
	Start Logical Test	
	Test Result Logical Test	OK
OPC Server Object	Start Test OPC Server	
	Test Result Object OPC Server	FAILED
OPC Group Object	Start Test Object OPC Group	
	Test Result Object OPC Group	OK
Performance Test	Start Test Performance	
	Test Result Performance Test	OK

Tabulka C.1: Výsledky OPC Compliance testu

Komponenta OPC server neprošla testem z důvodu vypuštěné implementace filtrování tagů adresního prostoru pomocí regulárních výrazů.

C.1.2 MATLAB

V prostředí MATLAB jsem testoval čtení/zápis dat. Vytvořil jsem dvě základní skupiny testů. V první z nich jsem využil „softwarové“ karty Advantech DEMO. Druhá skupina provádí podobné testy s fyzickou kartou Advantech PCI 1750. Jak už jsem zmínil dříve, to, co se zapíše na výstup softwarové karty se okamžitě objeví na vstupu. Pro oba případy jsem si v adresním prostoru vytvořil dva tagy - *Read* pro čtení vstupů a *Write* pro zápis na výstupy. Simulační dobu sem stanovil na 10s. Pro účely testování jsem vytvořil v nadstavbě MATLABu - simulinku, simulační schéma na obr. C.2.

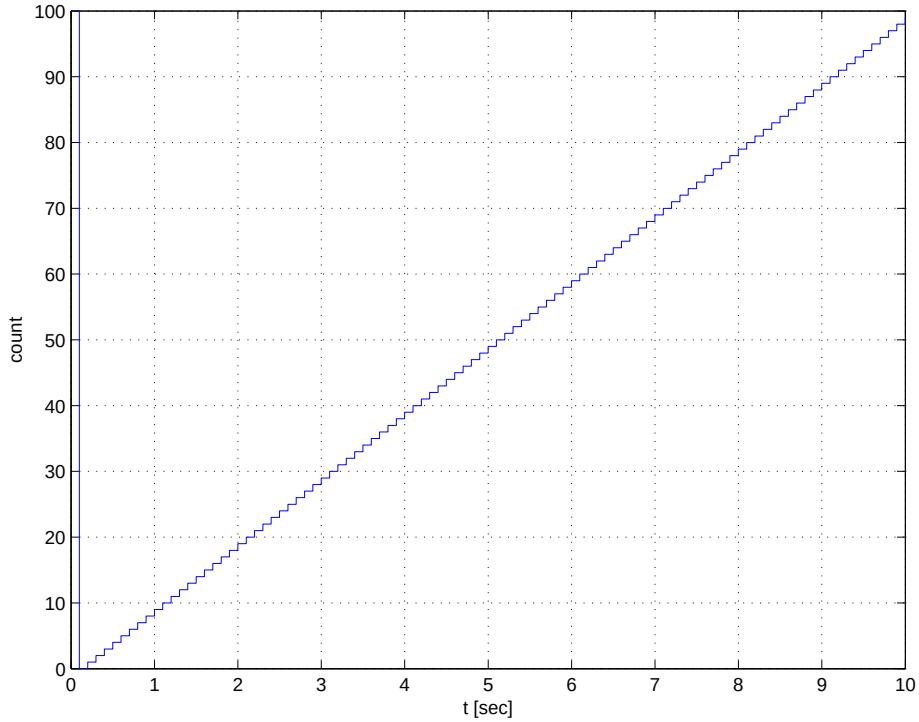


Obrázek C.2: Simulační schéma pro testy

C.1.2.1 Advantech DEMO

Do vstupů softwarové karty jsem zapojil 16 bitový čítač. Perioda zápisu je nejkratší možná, tedy 100ms. Shodná perioda je použita také pro čtení. Vyzkoušel jsem oba druhy, tedy asynchronní i synchronní čtení/zápis dat. Nejlépe dopadlo asynchronní čtení dat, které funguje velmi dobře bez ohledu na to, zda data byla zapsána synchronním čí asynchronním způsobem (obr. C.3). Stejný výsledek dává synchronní čtení dat se současnou aktualizací tagů v kombinaci se synchronním zápisem. Poněkud hůře je na tom synchronní

čtení dat se současnou aktualizací tagů v kombinaci s asynchronním zápisem a synchronní čtení dat bez aktualizace tagů s jakýmkoliv typem zápisu. Zde se stává, že se data v některých cyklech nestihnou aktualizovat (obr. C.4). Zpoždění je ale max. 0.1s, což v bude stačit při použití serveru na katedře řídící techniky.



Obrázek C.3: Výsledek asynchronního čtení

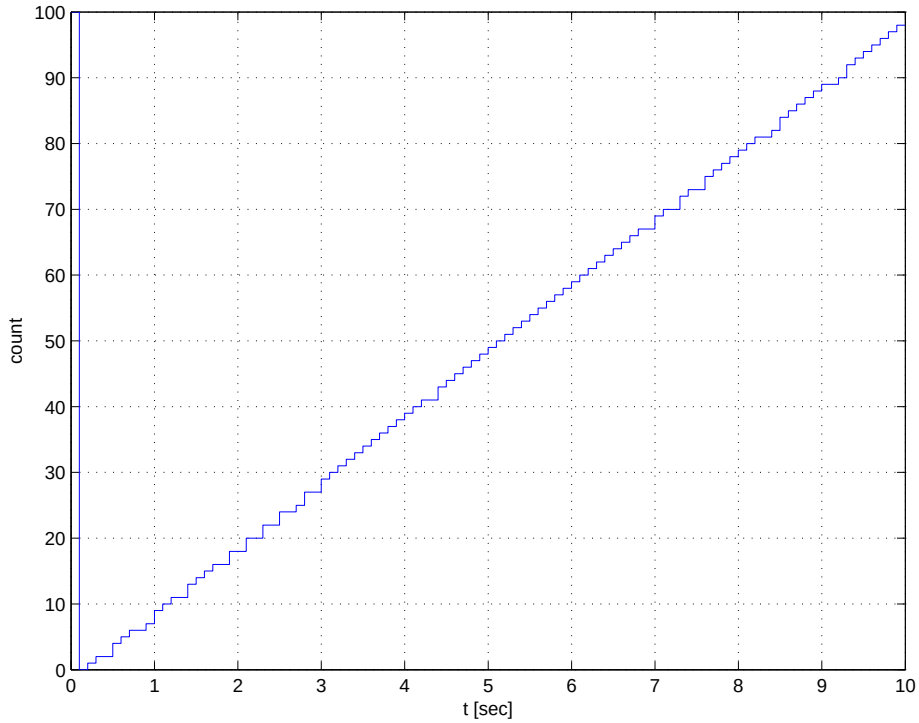
C.1.2.2 Advantech PCI 1750

Do fyzické karty Advantech jsem zapojil čítač poháněný generátorem o frekvenci 10Hz. Výsledná frekvence změn na výstupu čítače byla 5Hz a čítač pracoval v režimu modulo 4. Čtení hodnot proběhlo bez problémů. Na obr. C.5 je ukázka asynchronního čtení dat.

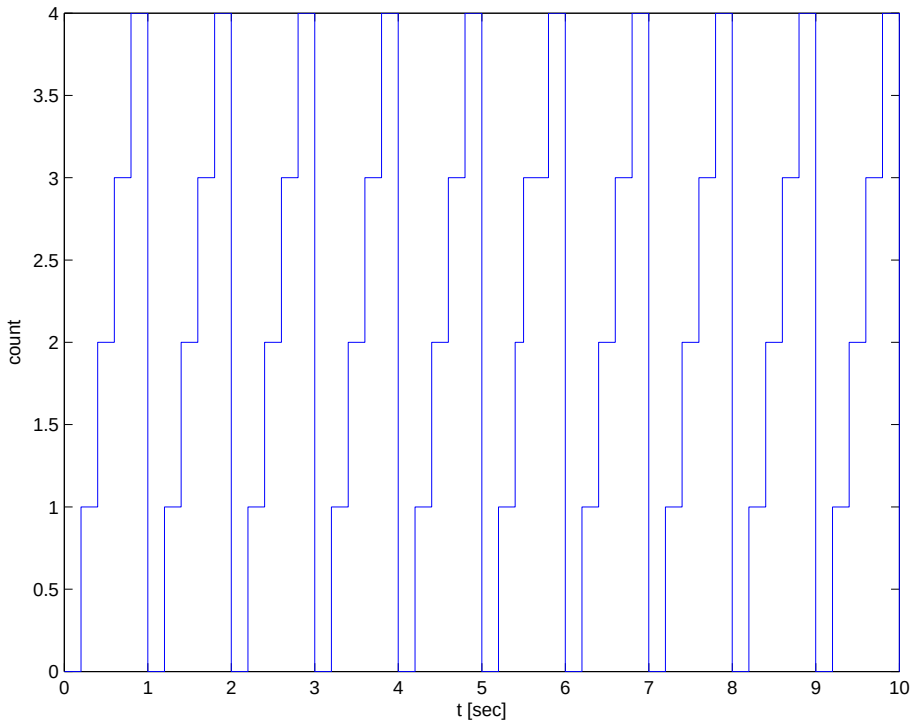
Zápis na kartu jsem prováděl taktéž pomocí 16 bitového čítače MATLABU. Kvalitu zápisu jsem kontroloval vizuálně pomocí připojených diod.

C.1.3 Ostatní klienti

Funkce serveru jsem dále prověřoval ve klientech MatrikonOPCExplorer, viz lit. [24] a Visual OPCTest Validator, viz lit. [25]. Zde sem provedl pouze jednoduché operace jako



Obrázek C.4: Výsledek synchronního čtení bez aktualizace tagů



Obrázek C.5: Výsledek asynchronního čtení z karty PCI 1750

přidání OPC skupiny a položky, s následným čtením/zápisem položek. Tyto testy měly bezproblémový průběh.

C.2 Klient a manažer

Tyto aplikace jsem testoval pouze několikanásobným proklikáním všech funkcí (viz ovládání v příloze B) a ruční simulací chybových stavů. Funkce klienta jsem ještě prověřil s volně dostupným serverem LightOPC, viz lit. [26].

Příloha D

CD-ROM, Instalace aplikací

Všechny vytvořené aplikace přikládám na CD. Pro jejich snadnou instalaci jsem vytvořil instalační program v prostředí Advanced Installer 3.9.

CD-ROM obsahuje

- adresář *Install* - obsahuje soubory setup.exe, DevMgr.exe, PCI1750.exe a OPC .NET API 1.30 SDK 3.00.msi.
- adresář *Source* - obsahuje adresáře se zdrojovými kódy aplikací, podstatné z nich popisuje kap. 6.
- adresář *DP* - obsahuje elektronickou verzi této práce.

Postup instalace

1. Pomocí DevMgr.exe nainstalovat základní ovladač pro zařízení Advantech.
2. Pomocí PCI1750.exe nainstalovat specifický ovladač pro kartu Advantech PCI 1750.
3. Pomocí OPC .NET API 1.30 SDK 3.00.msi nainstalovat knihovny pro OPC .NET API, nutné pro správné fungování klienta.
4. Pomocí setup.exe nainstaovat server, manažera a klienta.

Aplikace manažera a klienta vyžadují nainstalovaný .NET Framework 1.1. K instalaci je potřeba mít právo zápisu do registru Windows. Do nich se zapíší kromě klíčů pro zařízení Advantech a podpůrné knihovny, také tyto klíče nutné pro fungování serveru

- *HKEY_CLASSES_ROOT\CLSID\87F9169B-9390-44c9-9CA2-B206B09F9D94* - GUID pro ClassObject serveru

  - *ProgID* - název serveru přidělený pro GUID
  - *Implemented Categories* - obsahuje seznam GUID, který říká které verze specifikace OPC Data Access server podporuje
  - *LocalServer32* - obsahuje cestu k souboru serveru

- *HKEY_CLASSES_ROOT\OPC.AOPCSvr.1\CLSID* - klíč OPC.AOPCSvr.1 se shoduje s ProgID a je nutný pro správný překlad ProgID na GUID serveru.
- *HKEY_CLASSES_ROOT\AppID\87F9169B-9390-44c9-9CA2-B206B09F9D94* - identifikace serveru pro nastavení DCOM a používání serveru se vzdálených počítačů.

Po instalaci se v hlavním menu vytvoří nové submenu OPC Components, ze kterého lze spustit klienta nebo manažera serveru. Vlastní server se spouští automaticky na vyžádání klientem nebo manažerem. Odinstalování apliací je také velmi jednoduché. Stačí otevřít složku Přidat/Odebrat programy s ovládacích panelů Windows, najít v seznamu příslušné části a zvolit je k odinstalování. Záznamy v registrech jsou samozřejmě odebrány také.