České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra řídicí techniky

DIPLOMOVÁ PRÁCE

Klient analyzátoru protokolu Profibus DP

Praha, 2008 **Aleš Kautský**

Prohlášení

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

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

V Praze dne 20. května 2008

podpis

Poděkování

Děkuji především vedoucímu mojí diplomové práce Ing. Pavlu Burgetovi a konzultantovi Ing. Petru Smolíkovi za jejich cenné náměty, rady a připomínky, které mi velmi pomohly při vývoji aplikace popsané v této práci.

Dále děkuji mým rodičům a přítelkyni za neustálou psychickou podporu a zázemí během celého studia. Velké díky patří také mojí sestře za pomoc s jazykovou korekturou celé této práce.

Abstrakt

V mojí diplomové práci se zabývám implementací klientské aplikace pro analyzátor průmyslové sběrnice Profibus DP. Analyzátor je založen na architektuře **klient - server**. Komunikace mezi klientem a serverem probíhá po síti typu Ethernet. Server a ovladače k převodníkové kartě s RS-485 rozhraním jsou realizovány v paralelně probíhajících diplomových pracích mých kolegů.

Klientskou aplikaci vyvíjím na platformě .NET Framework 2.0 pro operační systémy Windows 2000/XP. Grafické rozhraní programu vytvářím s ohledem na přehlednost vizualizace dat. Pro snazší analýzu DP rámců je v programu k dispozici funkce pro filtraci a interpretaci zobrazovaných rámců. V programu umožňuji načítání data ze serveru a jejich následné uložení na lokální disk. Uložená data lze opětovně načíst a zobrazit. Pro komunikaci se serverem slouží aplikační protokol založený na XML.

Abstract

In my master thesis I deal with the implementation of a Profibus DP analysis tool, working as a client application in a client - server architecture. The communication between the client and the server is done via Ethernet. The server application and the RS-485 to PCI interface card is being developed by my colleagues in their master theses.

The client application is developed in .NET Framework 2.0 platform for Windows 2000/XP OS. The graphical user interface is oriented at data visualisation. The application allows easier analysis of the Profibus DP frames by applying certain functions concerning data filtration and interpretation. The data from the server can be saved in the local memory storage, and loaded and visualised later again. The communication protocol is based on the eXtended Markup Language (XML).

České vysoké učení technické v Praze - Fakulta elektrotechnická

Katedra řídicí techniky *Školní rok:* 2006/2007

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Aleš Kautský

Obor: Technická kybernetika

Název tématu: Klient analyzátoru protokolu Profibus DP

Zásady pro vypracování:

Realizujte klienta pod operačním systémem Windows 2000/XP na platformě .NET 2.0. Zpracujte grafické zobrazení přijatých zpráv v potřebném formátu a interpretujte jejich význam. Aplikujte offline filtr na přijaté zprávy.

1. Realizujte protokol komunikace se serverem.
2. Realizujte zobrazení zpráv vhodným způsobem, jejich filtrování a interpretaci dle normy Profibus DP.
3. Realizujte výpočet statistických údajů typu, jak často daná stanice komunikuje, kolik procent zabírá datová výměna, jak dlouhá je doba oběhu tokenu, apod.

Seznam odborné literatury: Dodá vedoucí práce.

Vedoucí diplomové práce: Ing. Pavel Burget

Termín zadání diplomové práce: zimní semestr 2006/2007 (změna zadání: 1.10.2007)

Termín odevzdání diplomové práce: leden 2008

L.S.

prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

doc. Ing. Boris Šimák, CSc.
děkan

V Praze dne 17.10.2007

Obsah

1 Úvod **1**

2 Profibus **3**

2.1 Fyzická vrstva 4

2.2 Linková vrstva 4

2.2.1 Řízení přístupu na komunikační médium 5

2.2.2 Struktura rámců 6

2.2.3 Služby linkové vrstvy 8

2.3 Aplikační a uživatelská vrstva 9

2.3.1 Uživatelská vrstva 9

2.4 Teoretické výpočty parametrů sběrnice Profibus 11

2.4.1 Výpočet t_{MC} v závislosti na přenosové rychlosti 11

2.4.2 Výpočet R_{SYS} v závislosti na přenosové rychlosti 13

2.4.3 Výpočet t_{SR} v závislosti na přenosové rychlosti 14

2.4.4 Výpočet t_{RR} v závislosti na počtu masterů 16

3 .NET Framework a struktura aplikace **19**

3.1 Platforma .NET 19

3.1.1 Překlad aplikace 20

3.1.2 MSIL 21

3.1.3 Jmenné prostory 21

3.2 Přenositelnost na Linux 21

3.2.1 MONO 22

3.2.2 DotGNU Portable.NET 22

3.3 Jazyk XML 22

3.4 Struktura aplikace 24

4 Nastavení aplikace **25**
4.1 Správa nastavení 25
4.1.1 Integrovaná správa nastavení 25
4.1.2 Vlastní správa nastavení 26
4.2 Jazyková lokalizace aplikace v .NETu 28
4.2.1 Lokalizace formulářů 29
4.2.2 Lokalizace dialogů 29

5 Grafické rozhraní aplikace **31**
5.1 Struktura hlavního okna aplikace 32
5.2 Upravené a nové komponenty 35
5.2.1 Komponenta MyLabel 35
5.2.2 Komponenta MyTitleBar 36
5.2.3 Komponenta MyButton 36
5.3 Položky nastavení 37
5.4 Formulář nastavení 37

6 Komunikace **39**
6.1 Popis datového záznamu 40
6.2 Vytvoření komunikačního kanálu 41
6.3 Komunikační protokol 44
6.3.1 Sémantika komunikačního protokolu 45
6.4 Implementace obsluhy komunikace 48
6.5 Přenos binárních dat 53
6.5.1 Formát dat na serveru 53
6.5.2 Řešení přenosu binárních dat 54

7 Zpracování a zobrazení dat **57**
7.1 Uchovávání dat v paměti počítače 57
7.2 Zpracování dat ze záznamu 59
7.3 Zpracování dat ze serveru 60
7.4 Zobrazení dat 61
7.4.1 Komponenta DataGridView 61
7.4.2 Interpretace dat podle specifikace Profibus DP 63
7.5 Filtrace a podmíněný záznam dat 64
7.5.1 Filtrace zobrazovaných dat 64

7.5.2 Filtrace a podmíněný záznam dat na serveru 66
7.6 Datové statistiky 67

8 Závěr **69**
8.1 Vylepšení do budoucna 70

A HW a SW požadavky **I**
A.1 Doporučený hardware I
A.2 Doporučený software II
A.2.1 SW pro spuštění aplikace II
A.2.2 SW pro překlad aplikace II

B DTD soubory komunikačního protokolu **III**
B.1 DTD komunikace od klienta k serveru. III
B.2 DTD komunikace od serveru ke klientovi IV

C Obsah přiloženého CD **V**

Seznam obrázků **V**

Seznam tabulek **VIII**

Seznam použitých zkratek **IX**

Literatura **XII**

Kapitola 1

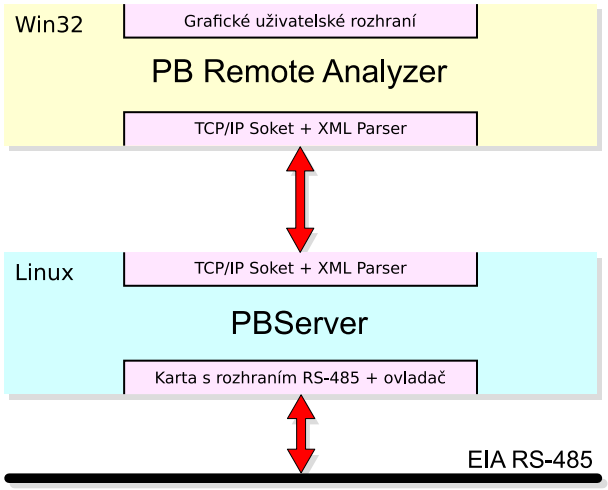
Úvod

V dnešní době je automatizace nedílnou součástí našich životů, ať se jedná o řídicí systém automatické pračky, světelné křižovatky či výrobní linky v automobilce. Zmíněné technologické procesy na první pohled vypadají velmi odlišně, ale mají společné rysy. Většina z nich je založena na distribuovaném řízení. Řídicí systém je rozdělen na moduly, které spolu komunikují pomocí komunikačních sítí. Tyto sítě, někdy označované jako sběrnice, musí pro optimální řízení procesu zajistit rychlou a bezporuchovou výměnu dat mezi jednotlivými částmi systému a zároveň rychlé a spolehlivé ovládání akčních členů. Vzdálenosti mezi bloky takového řídicího systému se pohybují v jednotkách metrů až kilometrů. Pro tyto účely bylo vyvinuto mnoho průmyslových sítí, které souhrnně označujeme názvem **Fieldbus**. Do této skupiny patří i sběrnice **Profibus** (*Process Field Bus*). Nejrozšířenější je varianta **DP** (*Decentralized Peripherals*) používaná hlavně v Evropě.

Technologické procesy bývají velmi rozsáhlé a při jejich uvádění do provozu i během běžného provozu se občas vyskytují nežádoucí chyby a výpadky komunikace. Uvedené výpadky mohou být způsobeny například stárnutím jednotlivých částí systému, nekompatibilitou zařízení od různých výrobců, mechanickým poškozením nebo elektromagnetický rušením. Zde nacházejí uplatnění analyzátory komunikace průmyslových sítí, pomocí nichž lze poměrně rychle odhalit zdroj chyby. Bez těchto analyzátorů by bylo velmi složité a v některých případech i nemožné nalézt příčinu výpadků komunikace.

Motivací mojí práce bylo vytvoření analyzátoru komunikace na sběrnici Profibus s možností dlouhodobého monitorování a ukládání dat. Analyzátor by měl dále poskytovat doplňkové funkce pro usnadnění analýzy dat.

Průmyslové prostředí je často agresivní, proto není vhodné pro umístění běžného počítače nebo laptopu. S ohledem na tuto skutečnost jsem zvolil koncepci analyzátoru založeného na architektuře **klient - server**. Schéma analyzátoru obsahuje obrázek 1.1.



Obrázek 1.1: Blokové schéma analyzátoru

Jako server může být použit počítač určený do průmyslového prostředí, který pomocí příslušného hardwarového vybavení čte a zpracovává data ze sběrnice Profibus. Data jsou dle volby ukládána lokálně nebo přímo odesílána do klientské aplikace přes TCP spojení.

Nespornou výhodou této koncepce analyzátoru je právě možnost vzdáleného přístupu k serveru. Lze se tedy z jednoho místa připojovat k různým serverům rozmístěným po technologii. Server může dlouhodobě ukládat data bez nutnosti připojení klientské aplikace. Vývojem serverové aplikace se zabývá Martin Samek ve své diplomové práci [1] s názvem „*Server analyzátoru protokolu Profibus DP*“.

Cílem mojí diplomové práce je návrh a implementace základních částí klientské aplikace sloužící jako rozhraní mezi vzdáleným serverem a uživatelem. Klientská aplikace je realizována na platformě .NET Framework pod operačním systémem Windows 2000/XP. Základním požadavkem na grafické rozhraní je přehledné zobrazení dat, jejich interpretace a možnost filtrace zobrazovaných rámců.

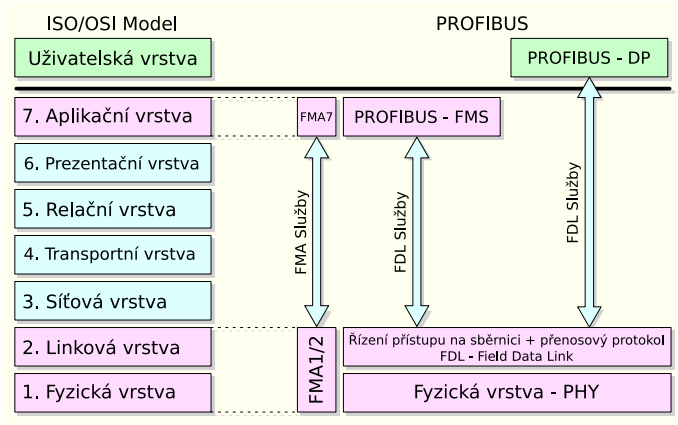
Výsledná verze klientské aplikace není rozhodně finální verzí určenou pro reálné použití. Výstup mojí diplomové práce bude sloužit jako základ dalších navazujících diplomových prací.

Kapitola 2

Profibus

Profibus DP se používá hlavně pro rychlou a spolehlivou komunikaci mezi řídicími jednotkami a decentralizovanými periferiemi. Řídicími jednotkami jsou nejčastěji programovatelné logické automaty (**PLC**) nebo průmyslové počítače. Periferie zastupují převážně vstupní/výstupní zařízení, senzory a akční členy.

Profibus je členěn do vrstev, které odpovídají referenčnímu modelu **ISO/OSI**. Následující obrázek 2.1 obsahuje schéma pro dva vybrané protokoly. **ISO/OSI** model definuje sedm vrstev. Profibus DP má navíc implementovánu uživatelskou vrstvu.



Obrázek 2.1: Porovnání s ISO/OSI modelem

FMS (*Fieldbus Message Specification*) je nejstarším protokolem Profibusu. Dnes se téměř nepoužívá a v podstatě ho nahrazuje verze DP. Oba protokoly sdílejí stejnou fyzickou a linkovou vrstvu a lze je provozovat společně v jedné síti. Protokol FMS obsahuje i aplikační vrstvu sloužící jako rozhraní mezi uživatelskou aplikací a linkovou vrstvou.

U protokolu DP je toto rozhraní implementováno až ve vrstvě uživatelské, která není součástí ISO/OSI modelu. Další vrstvy jsou vynechány z důvodu snížení ceny a zvýšení efektivity. V následujícím textu popisuji především funkci linkové a uživatelské vrstvy. Ty jsou důležité pro rozebírání obsahu rámců a jejich interpretaci dle protokolu DP.

2.1 Fyzická vrstva

Fyzická vrstva definuje elektrické a mechanické vlastnosti přenosového média a přístupového rozhraní. Norma Profibusu [2, Part 2] obsahuje více typů fyzických vrstev. Nejpoužívanějším standardem je EIA RS-485 s přenosovou rychlostí 9,6 kb/s – 12 Mb/s.

Topologie sítě je tvořena lineární sběrnicí zakončenou na obou koncích terminátory. Jako přenosové médium se používá stíněné dvouvodičové vedení. Maximální délka sběrnice je 1200 m při přenosových rychlostech $\leq 93{,}75$ kb/s. Pro vyšší rychlosti se délka vedení zkracuje. Maximální počet zařízení na vedení je omezen na 32. Pro prodloužení sběrnice a rozšíření počtu stanic existují speciální opakovače. Jejich prostřednictvím lze rozdělit síť do segmentů, které mají předchozí omezení, a vytvořit tak síť s hvězdicovitou topologií.

Pro přenos binární informace po médiu slouží NRZ (*Non Return to Zero*) kódování. Jeden z vodičů sběrnice je označen jako pozitivní a druhý jako negativní. Logické "1" odpovídá kladný rozdíl napětí mezi vodiči a logické "0" záporný rozdíl napětí.

V této práci se fyzickou vrstvou přímo nezabývám. Čtení dat ze sběrnice zajišťuje aplikace **PBServer** pomocí univerzální komunikační PCI karty s řadičem **OXmPCI954**. Podrobnosti o této kartě jsou k dispozici v diplomové práci Martina Samka [1], která popisuje vývoj zmiňované serverové aplikace.

2.2 Linková vrstva

Na obrázku 2.1 je linková vrstva Profibusu rozdělena na dvě části. Blok **FDL** (*Fieldbus Data Link*) se stará o řízení přístupu na sběrnici (**MAC** - *Medium Access Control*), sestavování vysílaných rámců (zpráv) a rozpoznávání přijatých rámců. Aplikační a uživatelské vrstvě poskytuje služby a funkce pro vysílání a příjem dat. Blok **FMA1/2** (Fieldbus Management Layer 1 and Layer 2) zasahuje až do fyzické vrstvy a zprostředkovává řízení a konfiguraci obou zmíněných vrstev.

2.2.1 Řízení přístupu na komunikační médium

Specifikace Profibusu umožňuje vytvoření komunikační sítě, která obsahuje až 127 zařízení. Každé zařízení má přidělenu unikátní adresu v rozsahu 0 – 126. Adresa 127 je vyhrazena pro rozesílání zpráv více stanicím v jednom okamžiku (Broadcast/Multicast).

Profibus DP rozlišuje následující typy zařízení:

Master třídy 1 – aktivní stanice, která řídí přístup na sběrnici a provádí cyklickou datovou výměnu se zařízeními typu slave. Dále může komunikovat i s masterem třídy 2. Do této kategorie patří především PLC.

Master třídy 2 – aktivní stanice, která poskytuje funkce pro diagnostiku a správu sítě. Umožňuje například acyklické čtení dat ze stanic typu slave, nastavování parametrů sběrnice a jednotlivých zařízení. Do této skupiny lze zařadit například vizualizační nebo diagnostický panel a případně i diagnostické PC s komunikační kartou.

Slave – pasivní stanice, která nemá možnost řídit přístup na sběrnici a pouze odpovídá na požadavky libovolného zařízení typu master.

Pro běžné účely je počet stanic typu master v komunikační síti omezen na 32. Pro časově nekritické aplikace lze připojit až 127 masterů. V případě protokolu DP se doporučuje nepřipojovat více jak 3 stanice master do jedné sítě.

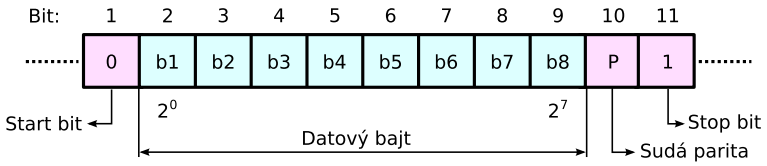
Profibus je založen na hybridním přístupu na médium, který kombinuje *Token Passing* a *Master-Slave* komunikaci. Aktivní stanice spolu tvoří logický kruh, ve kterém si předávají pověření (*token*). Stanice master, která drží token může komunikovat se slave zařízeními po dobu danou T_{TH} (*Token Holding Time*). Po uplynutí této doby master předá pověření další aktivní stanici s vyšší adresou. Aktivní stanice s nejvyšší adresou předává pověření zpět stanici s nejnižší adresou a celý proces se opakuje. Pokud je na síti jen jeden master, tak předává token sám sobě. Při inicializaci sítě začne vždy jako první komunikovat aktivní stanice s nejnižší adresou.

Každá stanice master si udržuje dva seznamy adres: **LAS** (*List of Active Stations*) a tzv. **GAP list**. LAS obsahuje adresy všech aktivních stanic připojených v síti. Každá aktivní stanice si tento seznam vytváří ještě před svým zařazením do logického kruhu, kdy pouze naslouchá komunikaci na sběrnici. Seznam GAP obsahuje informace o obsazenosti adresního prostoru. Každý master si udržuje GAP list v rozsahu od vlastní adresy do adresy další stanice master. Oba seznamy se cyklicky obnovují. Díky tomuto bezkoliznímu deterministickému řízení přístupu na médium je možné připojovat a odpojovat zařízení i za provozu.

Další podrobnosti o komunikaci a řešení krizových stavů na sběrnici (ztráta tokenu, více tokenů, připojení a odpojení zařízení...) lze nalézt ve specifikaci sběrnice Profibus [2, Part 4]. Jako zdroj informací lze také použít diplomovou práci Pavla Trnky [3] a Jana Blepty [4], kteří se zabývali implementací stanice master na PC.

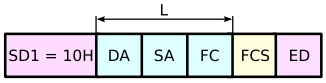
2.2.2 Struktura rámců

Mezi fyzickou a linkovou vrstvou se předávají posloupnosti bitů. Při příjmu dat FDL vrstva skládá z těchto bitů tzv. UART znak (viz obrázek 2.2) a následně z těchto znaků složí rámce. Při odesílání dat je postup opačný. Každý znak je dlouhý 11 bitů. *Start bit* a *Stop bit* slouží pro synchronizaci asynchronního přenosu dat a vždy mají uvedené logické hodnoty. Bity **b1** – **b8** jsou informační a z jejich obsahu je vypočtena sudá parita **P**.

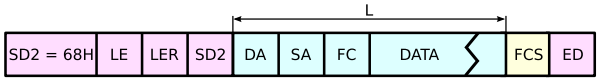


Obrázek 2.2: Struktura UART znaku

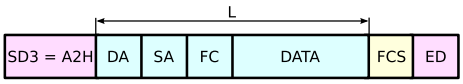
Specifikace Profibusu rozlišuje následujících pět formátů rámců. Protokol DP nepoužívá rámec s pevnou délkou datové sekce uvedený na obrázku 2.5. Potvrzovací rámec na obrázku 2.7 slouží jako odpověď na požadavky, kde není nutné předávat data.



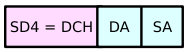
Obrázek 2.3: Rámec pevné délky bez datové sekce



Obrázek 2.4: Rámec s proměnnou délkou datové sekce



Obrázek 2.5: Rámec s pevnou délkou datové sekce



Obrázek 2.6: Rámec pro předávání tokenu

Obrázek 2.7: Potvrzovací rámec (*Short Acknowledge*)

Zkratky uvnitř jednotlivých rámců mají následující význam:

SD – *Start Delimiter*. Podle hodnoty tohoto znaku se rozlišují formáty rámců.

DA – *Destination Address*, adresa cílové stanice.

SA – *Source Address*, adresa zdrojové stanice.

FC – *Frame Control*, obsahuje informace o typu zprávy (požadavek/odpověď), funkce nebo služby FDL a u zpráv odpovědí lze zjistit i typ a FDL status odpovídající stanice.

FCS – *Frame Checksum*, slouží jako kontrolní součet informačního pole označeného písmenem L.

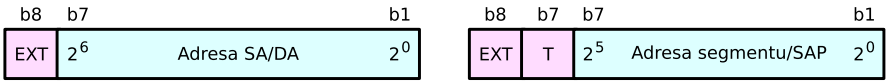
LE – *Length*, obsahuje počet znaků informačního pole L u rámce s proměnnou délkou.

LER – Tento znak by měl mít stejnou hodnotu jako LE.

ED – *End Delimiter*, označuje konec rámce. Má vždy hodnotu 16H.

Sekce DATA u rámce s pevnou délkou 2.5 obsahuje právě 8 datových bajtů. Pokud je informace kratší, tak se zbylé bajty nastaví na 0. V rámci s proměnnou délkou 2.4 je možné přenášet 1 – 246 datových bajtů.

Datová sekce může na svém začátku obsahovat rozšíření cílové adresy (**DAE** - *Destination Address Extension*) a/nebo zdrojové adresy (**SAE** - *Source Address Extension*). Pokud se vyskytují obě rozšíření zároveň, pak je jejich pořadí stejné jako u adres. Obrázek 2.8 ilustruje vlevo strukturu bajtu adresy a vpravo strukturu bajtu rozšířené adresy.



Obrázek 2.8: Formát adresy a rozšířené adresy

O existenci rozšířené adresy rozhoduje bit EXT v DA a SA. Pokud EXT = 1, pak datová sekce obsahuje rozšířenou adresu DAE respektive SAE. Zmíněné rozšíření může mít délku jeden nebo dva znaky, které mají formát dle obrázku 2.8 vpravo. O délce rozhoduje bit EXT v prvním znaku rozšíření. Bit T určuje typ rozšířené adresy. Pokud T = 1, pak se jedná o adresu segmentu a pro T = 0 je rozšířená adresa označena jako přístupový bod služby - **SAP** (*Service Access Point*). Rozsah SAPu je 0 – 62 u zdrojového přístupového bodu (**SSAP**) a 0 – 63 u cílového přístupového bodu (**DSAP**). U protokolu Profibus DP je segmentová adresa ignorována a uvažují se pouze SAPy.

Každá stanice má ve své FDL vrstvě aktivované určité přístupové body. Některé mají přiřazenu určitou funkci podle používaného komunikačního protokolu. Při komunikaci se SAPy vkládají do zmíněného rozšíření adres rámce a lze tak identifikovat typ a funkci zprávy daného protokolu (např. Profibus DP). U rámců, kde nejsou SAPy uvedeny, se předpokládá použití tzv. **Default SAPu**, který se také označuje jako **NIL**. Popisem identifikace typů funkcí se zabývám v kapitole o uživatelské vrstvě 2.3 a dále pak v kapitole o interpretaci dat rámce podle protokolu Profibus DP 7.4.2.

2.2.3 Služby linkové vrstvy

V předchozím textu jsem naznačil, že komunikace mezi linkovou vrstvou a vyššími vrstvami probíhá pomocí služeb. V této kapitole uvádím jejich výčet s krátkým popisem. U každé z následujících služeb lze nastavit prioritu *High* (vysoká priorita) nebo *Low* (nízká priorita). Protokol Profibus DP využívá pro přenos dat pouze služby SRD a SDN.

SDA – *Send Data with Acknowledge*, služba umožňuje poslat data jedné vzdálené stanici, která následně potvrdí příjem. Při chybě se přenos může opakovat.

SDN – *Send Data with No Acknowledge*, služba umožňuje poslat zprávu jedné vzdálené stanici, více vzdáleným stanicím (Multicast) nebo všem vzdáleným stanicím (Broadcast). Potvrzení zprávy ze strany příjemce ani opakování přenosu při chybě se neprovádí.

SRD – *Send and Rquest Data with Reply*, služba umožňuje odeslat data jedné vzdálené stanici a zároveň přijmout data z této stanice. V případě chyby je možné opakovat přenos.

2.3 Aplikační a uživatelská vrstva

Protokoly aplikační a uživatelské vrstvy dávají význam rámcům z linkové vrstvy. Na obrázku 2.1 je aplikační vrstva Profibusu rozdělena na dvě části. Blok **FMA7** (*Field Management Layer 7*) poskytuje uživatelské aplikaci služby pro řízení komunikace na sběrnici a správu nižších vrstev. Blok FMS zjednodušuje přístup k linkové vrstvě pro aplikace využívající protokol FMS. Pro analyzátor protokolu DP není aplikační vrstva důležitá, proto ji dále nepopisuji. Podrobnosti o zmiňovaných blocích obsahuje specifikace Profibusu [2, Part 5 – Part 7].

2.3.1 Uživatelská vrstva

Uživatelská vrstva na obrázku 2.1 specifikuje zprávy protokolu Profibus DP. Každá zpráva má přesně definovanou strukturu a význam dat. Uvnitř uživatelské vrstvy se nachází také blok s názvem **DDLM** (*Direct Data Link Mapper*), který má pro každou zprávu protokolu DP předdefinovanou funkci. Uživatelská aplikace volá přímo tyto funkce a předává jim data. DDLM pak podle typu funkce zvolí vhodnou FDL službu a doplní další parametry, ze kterých se na úrovni linkové vrstvy složí rámec. Proces pracuje i opačně v případě příjmu zprávy.

V aplikaci jsem implementoval obdobu bloku DDLM. Data ze serveru jsou zpracována na jednotlivé rámce. U nich je následně zkontrolována struktura (viz sekce 2.2.2) a podle tabulky 2.1 identifikován typ zprávy protokolu DP.

Typ DP zprávy	**SSAP**	**DSAP**	**Služba FDL**	**Priorita**
Komunikace Master - Slave				
Data_Exchange.req	nil	nil	SRD	high
Data_Exchange.res	nil	nil		low/high
Chk_Cfg.req	62	62	SRD	high
Set_Prm.req	62	61	SRD	high
Slave_Diag.req	62	60	SRD	high
Slave_Diag.res	60	62		low
Get_Cfg.req	62	59	SRD	high
Get_Cfg.res	59	62		low
Global_Control.req	62	58	SDN	high
RD_Outp.req	62	57	SRD	high
RD_Outp.res	57	62		low
RD_Inp.req	62	56	SRD	high
RD_Inp.res	56	62		low
Komunikace Master - Master				
.req	54	54	SRD	low
.res	54	54		low
Set_Slave_Add.req	62	55	SRD	high
Act_Para_Brct.req	54	54	SDN	low

Tabulka 2.1: Identifikace typů zpráv podle SAPů a FDL služeb.

První sloupec obsahuje název DP zprávy. Stejné názvy jsou použity i v popisu DDLM vrstvy ve specifikaci Profibusu [2, Part 8]. Koncovkou .req a .res označuji, zda se jedná o rámec požadavku nebo odpovědi. Další sloupce obsahují zdrojový a cílový přístupový bod služby. V posledních dvou sloupcích se nalézá typ a priorita FDL služby.

Zprávy lze rozdělit do dvou skupin. První skupina obsahuje zprávy pro komunikaci mezi stanicí master a slave. Druhá skupina obsahuje zprávy pro komunikaci mezi stanicí master třídy 1 a master třídy 2. Mezi mastery stejné třídy se žádné zprávy nepředávají.

Problematika interpretace dat obsažených ve zprávách je značně rozsáhlá. Kompletní a velmi podrobný popis obsahuje specifikace Profibusu [2, Stránka 750 – 779], proto ho zde neuvádím.

2.4 Teoretické výpočty parametrů sběrnice Profibus

V následujícím textu jsem určil mezní hodnoty některých parametrů komunikace na sběrnici Profibus. Mezi tyto parametry patří délka *Message Cyklu* (**MC**), počet MC vykonaných za sekundu a reakční čas systému. Výpočet provádím proto, abych získal představu o množství zpráv zpracovávaných v klientské aplikaci. Ve výpočtech používám minimální hodnoty zpoždění a neuvažuji zprávy *FDL Status Request*, pomocí nichž si stanice master obnovují obsah svého adresního prostoru (GAP list). Díky těmto zjednodušením jsem získal hodnoty parametrů, které v reálném systému nejsou pravděpodobně dosažitelné, ale jako odhad mezních hodnot by měly stačit. Především při vyšších přenosových rychlostech sběrnice lze očekávat rozdíl odhadnutých a skutečných hodnot parametrů 20 – 30 %. Pro výpočty jsem použil vzorečky ze specifikace linkové vrstvy Profibusu [2, Part 4].

2.4.1 Výpočet t_{MC} v závislosti na přenosové rychlosti

t_{MC} – (*Message Cycle Time*) udává čas potřebný k provedení jedné datové výměny mezi jednou stanicí master a jednou stanicí slave. MC se tedy skládá z rámce požadavku a z rámce odpovědi. Délka trvání jednoho MC lze následně vypočítat z času potřebného pro přenos rámců, ze zpoždění při přenosu rámců a ze zpoždění jednotlivých stanic.

$$T_{MC} = T_{S/R} + T_{SDR} + T_{A/R} + T_{ID} + 2 \cdot T_{TD} \quad \text{[bit]}. \tag{2.1}$$

Tento vzorec však neudává délku MC v jednotkách času, ale v bitech. Převod na čas je velmi jednoduchý. Stačí T_{MC} vynásobit převrácenou hodnotou přenosové rychlosti.

$$t_{MC} = T_{MC} \cdot t_{BIT} \quad \text{[s]}. \tag{2.2}$$

Jednotlivé proměnné ve vzorcích (2.1) a (2.2) mají následující význam:

t_{BIT} – (*Bit Time*) je roven převrácené hodnotě přenosové rychlosti.

$T_{S/R}$, $T_{A/R}$ – specifikují dobu potřebnou pro přenos jednotlivých rámců požadavků (*Send/Request*) a rámců odpovědí (*Acknowledge/Response*). Jelikož každý UART znak rámce má 11 bitů, lze tyto parametry určit jako: $T_{S/R} = a \cdot 11$ bit, kde a je počet znaků rámce. Pro $T_{A/R}$ je postup stejný.

T_{SDR} – (*Station Delay Time*) udává čas potřebný k dekódování přijatého rámce a sestavení vysílaného rámce odpovědi. Tento parametr je závislý na konkrétní implementaci protokolu v dané stanici a jako jeho minimální hodnota se obecně uvažuje $minT_{SDR} = 11$ bit.

T_{TD} – (*Transmission Delay Time*) udává maximální zpoždění na přenosovém médiu mezi vysláním rámce na straně vysílající stanice a přijetím rámce na straně příjemce.

T_{ID} – (*Idle Time*) specifikuje dobu, po kterou je na sběrnici klid mezi přijetím odpovědi a vysláním dalšího rámce požadavku. Hodnotu T_{ID} jsem určil podle vzorce:

$$T_{ID} = \max\{T_{SYN} + T_{SM}, \min T_{SDR}, T_{SDI}\} \quad [\text{bit}], \tag{2.3}$$

T_{SYN} – (*Synchronization Time*) udává minimální časový interval sloužící k synchronizaci stanic připojených na sběrnici. Obvyklá hodnota je $T_{SYN} = 33$ bit.

T_{SM} – (*Safety Margin*) význam tohoto parametru není ve specifikaci přesně popsán. Jako běžná hodnota se uvádí $T_{SM} = 4$ bity.

T_{SDI} – (*Station Delay of Initiator*) zpoždění vysílací stanice při zpracování rámců.

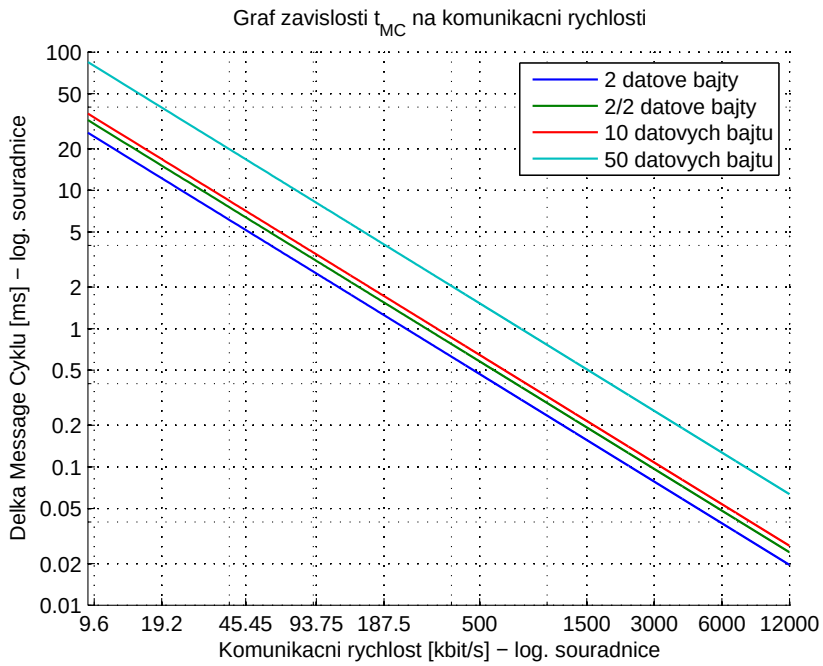
Následující tabulka 2.2 obsahuje vypočtené hodnoty t_{MC} v závislosti na přenosové rychlosti a na velikosti přenášených dat. Výpočet jsem provedl podle výše uvedených vzorců (2.1) a (2.2), do kterých jsem dosadil parametry: $T_{SDR} = minT_{SDR} = 11$ bitů; $T_{SYN} = 33$ bitů; $T_{SM} = 4$ bity; $T_{ID} = 37$ bitů; $T_{TD} = 0$ bitů. Poslední parametr signalizuje zanedbání zpoždění při přenosu dat po sběrnici. Dále jsem zanedbal opakování zpráv, acyklickou komunikaci a předávání tokenu, protože uvažuji komunikační síť pouze s jednou stanicí master.

Rychlost [kb/s]		9,6	19,2	45,45	93,75	187,5	500	1500	3000	6000	12000
Data	2 bajty	24,48	12,24	5,17	2,51	1,25	0,47	0,157	0,078	0,0392	0,0196
	2/2 bajty	30,21	15,10	6,38	3,09	1,55	0,58	0,193	0,097	0,0483	0,0242
	10 bajtů	33,65	16,82	7,11	3,45	1,72	0,65	0,215	0,108	0,0538	0,0269
	50 bajtů	79,48	39,74	16,79	8,14	4,07	1,53	0,509	0,254	0,1272	0,0636

Tabulka 2.2: Závislost t_{MC} [ms] na přenosové rychlosti a velikosti dat.

První řádek v tabulce je vypočten pro případ, kdy rámec požadavku tj. zpráva zaslaná stanicí master stanici slave neobsahuje žádná data (viz formát rámce na obrázku 2.3). Jako odpověď od stanice slave následně přijde rámec s proměnnou délkou datového pole

(viz formát rámce na obrázku 2.4). Pro první řádek je délka datového pole 2 znaky. Ve třetím a čtvrtém řádku tabulky jsou použity stejné rámce, pouze odpovědi mají datové pole o délce 10 a 50 znaků. Ve druhém řádku tabulky uvažuji případ, kdy si obě strany předávají rámce s datovým polem o délce 2 znaky, tzn. i stanice master posílá data stanici slave. Pro úplnost připojuji ještě graf závislosti v logaritmických souřadnicích.



Obrázek 2.9: Graf závislosti délky *Message Cyklu* na přenosové rychlosti.

2.4.2 Výpočet R_{SYS} v závislosti na přenosové rychlosti

R_{SYS} – (*Message Rate*) odpovídá počtu MC, které mohou proběhnout za sekundu.

Výpočet R_{SYS} je velice jednoduchý. Stačí vzít čas t_{MC} a vypočítat jeho převrácenou hodnotu.

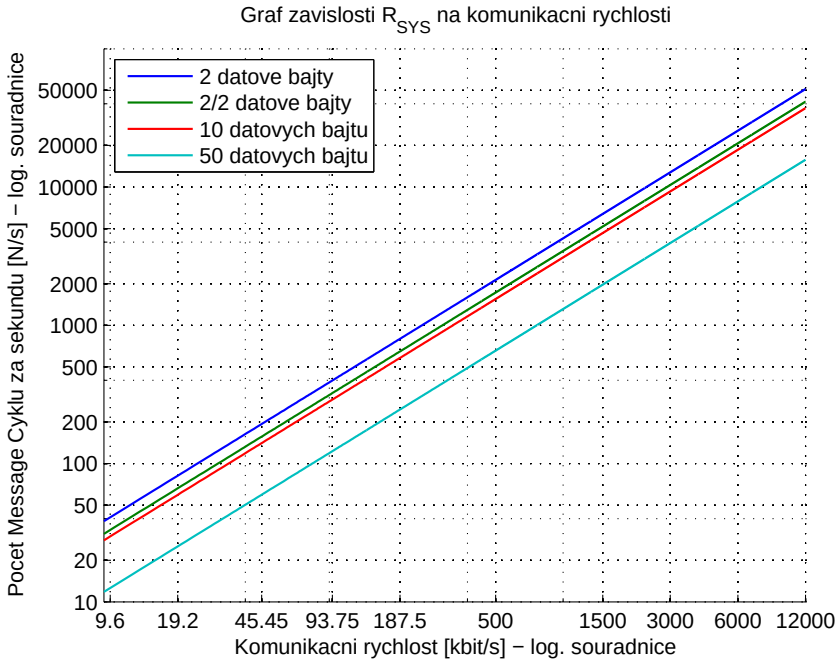
$$R_{SYS} = 1/t_{MC} \quad [\mathrm{N/s}]. \tag{2.4}$$

Následující tabulka 2.3 obsahuje hodnoty R_{SYS}, přičemž délky datových polí a komunikační rychlosti jsou uvažovány stejné jako v případě výpočtu t_{MC}. I v tomto případě uvažuji síť s jednou stanicí master a jednou stanicí slave. Opakování vysílání zpráv acyklickou komunikaci a předávání tokenu opět zanedbávám.

Rychlost [kb/s]		9,6	19,2	45,45	93,75	187,5	500	1500	3000	6000	12000
Data	2 bajty	41	82	193	399	798	2128	6383	12766	25532	51064
	2/2 bajty	33	66	157	323	647	1724	5172	10345	20690	41379
	10 bajtů	30	59	141	290	580	1548	4644	9288	18576	37152
	50 bajtů	13	25	60	123	246	655	1966	3932	7864	15727

Tabulka 2.3: Závislost R_{SYS} [N/s] na přenosové rychlosti a velikosti dat.

Výsledné charakteristiky zobrazuje graf závislosti v logaritmických souřadnicích.



Obrázek 2.10: Graf závislosti počtu MC za sekundu na přenosové rychlosti.

2.4.3 Výpočet t_{SR} v závislosti na přenosové rychlosti

t_{SR} – (*System Reaction Time*) v systému s jednou stanicí master a `n` stanicemi slave odpovídá tento parametr času, který je potřebný k proběhnutí jedné datové výměny mezi stanicí master a každou stanicí slave.

Hodnotu T_{SR} lze určit následujícím způsobem pomocí délky MC:

$$T_{SR} = \mathrm{n} \cdot T_{MC} + \mathrm{m} \cdot retT_{MC} \quad [\mathrm{bit}]. \tag{2.5}$$

Výstup z předchozího vzorce je možné převést na čas podobně jako v případě T_{MC}. Stačí výsledek v bitech vynásobit převrácenou hodnotou přenosové rychlosti (viz (2.2)).

Parametry v rovnici (2.5) mají následující význam:

`n` – udává počet stanic typu slave, se kterými komunikuje stanice master.

T_{MC} – délka trvání *MC* spočítaná podle (2.1).

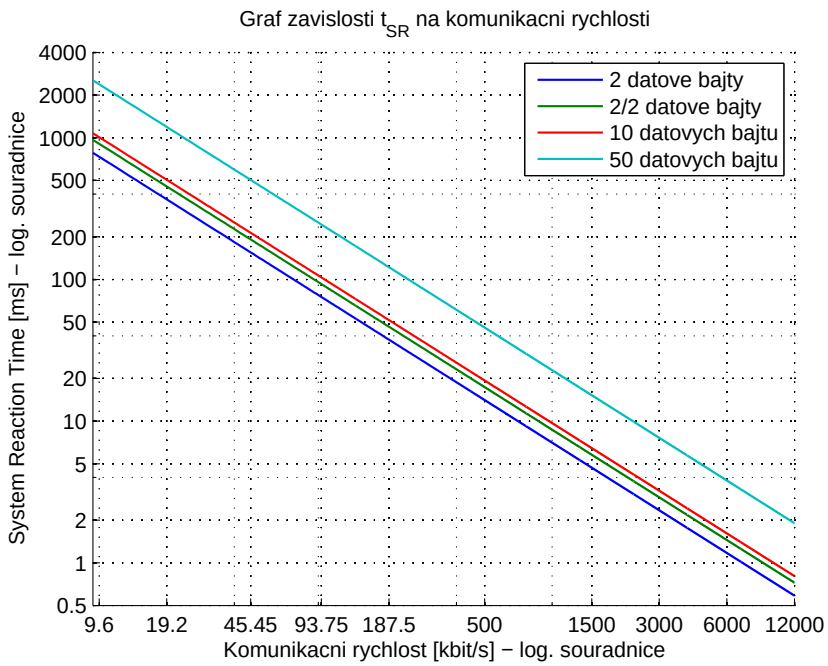
`m` – počet opakovaných zpráv během jednoho cyklu sběrnice. Cyklus sběrnice označuje proběhnutí jedné datové výměny mezi stanicí master a každou stanicí slave, která je k ní přiřazena.

$retT_{MC}$ – (*Message Retry Cycle Time*) udává délku MC při opakovaném posílání zprávy po vzniku chyby.

Následující tabulka 2.4 obsahuje výsledky výpočtů provedených podle vzorce (2.5). Pro lepší přehlednost jsem opět hodnoty převedl na čas podle vzorce (2.2). Parametry jsem nastavil na 30 stanic typu slave (`n = 30`) a žádné opakování zpráv (`m = 0`). Stejně jako v předchozích případech jsem neuvažoval předávání tokenu a acyklickou komunikaci.

Rychlost [kb/s]		9,6	19,2	45,45	93,75	187,5	500	1500	3000	6000	12000
Data	2 bajty	734	367	155	75	38	14,1	4,7	2,35	1,175	0,588
	2/2 bajty	906	453	191	93	46	17,4	5,8	2,90	1,450	0,725
	10 bajtů	1009	505	213	103	52	19,4	6,5	3,23	1,615	0,808
	50 bajtů	2384	1192	504	244	122	45,8	15,3	7,63	3,815	1,908

Tabulka 2.4: Závislost t_{SR} [ms] na přenosové rychlosti a velikosti dat.



Obrázek 2.11: Graf závislosti t_{SR} na přenosové rychlosti pro 30 stanic typu slave.

V případě sítě s více stanicemi master je maximální hodnota systémového reakčního času t_{SR} závislá na t_{RR} (*Real Token Rotation Time*), který je popsán v následující části. V podstatě k času t_{SR} systému s jednou stanicí master přibude čas potřebný k předávání tokenu.

2.4.4 Výpočet t_{RR} v závislosti na počtu masterů

t_{RR} – (*Real Token Rotation Time*) v komunikační síti s více stanicemi master odpovídá tento parametr času, který je potřebný k oběhu tokenu přes všechny stanice master. Každý master provede vždy datovou výměnu se svými stanicemi slave. V následující části jsem se pokusil odhadnout mezní hodnotu t_{RR} v závislosti na počtu stanic master, který se pohybuje v rozmezí 2 – 30. Vzal jsem v úvahu, že každá stanice master po obdržení tokenu provede vždy jen jednu datovou výměnu s jednou stanicí slave. Velikost předávaného datového pole je 2 znaky v obou směrech. Stejně jako v předchozích případech jsem zanedbal opakované posílání zpráv a zpoždění přenosu dat po sběrnici.

K výpočtu hodnot T_{RR} je nutné znát čas potřebný k předávání tokenu T_{TC} (*Token Cycle Time*)

$$T_{TC} = T_{TF} + T_{TD} + T_{ID} \quad \text{[bit]}. \tag{2.6}$$

T_{TF} – (*Token Frame Time*) udává čas potřebný k poslání tokenu. Podle obrázku 2.6 je délka rámce pro předávání tokenu rovna 3, a proto $T_{TF} = 33$ bitů.

T_{TD}, T_{ID} – význam těchto časů a jejich hodnoty jsem uvedl u (2.1).

Pro určení času jedné datové výměny mezi stanicemi a následné předání tokenu další aktivní stanici jsem dospěl k rovnici

$$T_{TC/MC} = T_{MC} + T_{TC} \quad \text{[bit]}. \tag{2.7}$$

Z toho pro `r` stanic master plyne jednoduchý vztah pro T_{RR}

$$T_{RR} = \mathrm{r} \cdot T_{TC/MC} \quad \text{[bit]}. \tag{2.8}$$

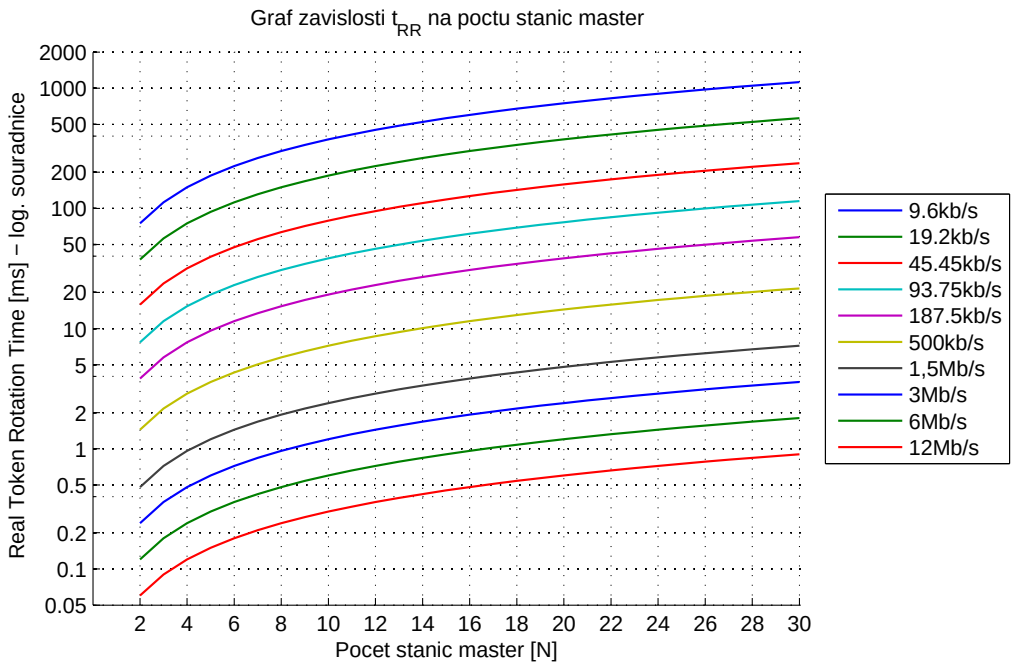
Převod na jednotky času jsem opět provedl podle (2.2). Vypočítané hodnoty obsahuje následující tabulka 2.5.

Rychlost [kb/s]		9,6	19,2	45,45	93,75	187,5	500	1500	3000	6000	12000
Počet stanic master	2	75	38	16	8	3,84	1,44	0,48	0,24	0,12	0,06
	3	113	56	24	12	5,76	2,16	0,72	0,36	0,18	0,09
	4	150	75	32	15	7,68	2,88	0,96	0,48	0,24	0,12
	5	188	94	40	19	9,60	3,6	1,2	0,6	0,3	0,15
	10	375	188	79	38	19	7,2	2,4	1,2	0,6	0,3
	15	563	281	119	58	29	10,8	3,6	1,8	0,9	0,45
	20	750	375	158	77	38	14,4	4,8	2,4	1,2	0,6
	25	938	469	198	96	48	18	6	3	1,5	0,75
	30	1125	563	238	115	58	21,6	7,2	3,6	1,8	0,9

Tabulka 2.5: Závislost t_{RR} [ms] na počtu stanic master a přenosové rychlosti.

Hodnota t_{RR} není nikde přímo zadávána a každá stanice master si tento čas měří sama. Přímo zadaná je pouze hodnota času t_{TR} (*Target Rotation Time*), což je v podstatě horní mez pro t_{RR}. Princip činnosti je následující: po obdržení tokenu stanice master nastaví do

časovače (*Token Rotation Timer*) T_{TR}. Tato hodnota se s časem dekrementuje a při dalším přijetí tokenu master přečte zbývající čas a přenastaví časovač opět na t_{TR}. Zbývající čas je v podstatě doba, po kterou může stanice master přistupovat na sběrnici než předá token dále. Označuje se jako T_{TH} (*Token Holding Time*). T_{TH} lze určit jako $T_{TR} - T_{RR}$. Při špatném nastavení parametrů sběrnice se může stát, že hodnota T_{TR} bude příliš malá a stanici master již při přijetí tokenu nebude zbývat čas pro komunikaci. V tomto případě je umožněno, aby master odeslal jednu zprávu s vysokou prioritou. Z toho plyne, že zprávy s nízkou prioritou je možné posílat pouze v případě, kdy bude platit $T_{RR} < T_{TR}$.



Obrázek 2.12: Graf závislosti t_{RR} na počtu masterů a přenosové rychlosti.

Kapitola 3

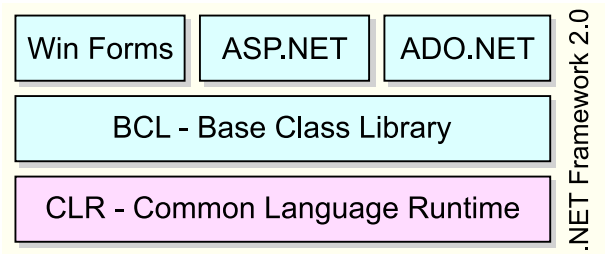
.NET Framework a struktura aplikace

V této kapitole se zabývám základním popisem platformy **.NET Framework** a jejími alternativami pro operační systém (**OS**) Linux. Další část kapitoly obsahuje základy jazyka **XML** (*Extensible Markup Language*), který používám pro ukládání nastavení a jazykovou lokalizaci aplikace. V poslední části kapitoly popisuji strukturu programu a jeho blokové schéma.

3.1 Platforma .NET

Za posledních několik let se .NET Framework stal hlavní vývojovou platformou pro operační systémy Windows. Pro tuto práci jsem zvolil verzi 2.0 zaručující spustitelnost aplikace v OS Windows 98/2000/XP/2003/Vista.

.NET tvoří rozsáhlou knihovnu funkcí pro vývojáře a zároveň slouží jako běhové prostředí pro aplikace. Strukturu obsahuje obrázek 3.1. Knihovna základních tříd - **BCL** (*Base Class Library*) poskytuje kolekci tříd založených na tzv. spravovaném kódu. Tyto třídy umožňují vykonávat prakticky všechny úlohy, které byly dříve prováděny pouze přes **Windows API** (*Application Programming Interface*). Windows API samozřejmě funguje i nadále a .NET lze chápat jako nadstavbu ulehčující přístup k funkcím OS. Další bloky nad BCL obsahují knihovny pro tvorbu oken aplikací (**Windows Forms**), internetových stránek (**ASP.NET**) a přístup k datům a jejich zpracování (**ADO.NET**).



Obrázek 3.1: Blokové schéma platformy .NET

Pojmem spravovaný kód je myšlen kód jazyka určeného pro platformu .NET. Do této skupiny patří především jazyk C#, jež byl od základu vyvinut pro tuto platformu. Dále je možné použít jazyky C++ a Visual Basic, které byly upraveny a rozšířeny o možnost psaní spravovaného kódu. V této práci jsem použil pouze C#. Tento jazyk vychází z C a C++, ale proti nim je daleko intuitivnější, srozumitelnější a především plně objektový. Jazyk C++ obsahuje velké množství knihoven, a proto je velmi složité se v něm vyznat. Na druhou stranu C++ je vhodnější pro časově náročné (*Real Time*) aplikace. Pro tyto účely se hodí nespravovaná verze jazyka využívající přímo Windows API.

Velmi důležitým blokem na obrázku 3.1 je **CLR** (*Common Language Runtime*), který slouží jako výše zmiňované běhové prostředí pro aplikace. Provádí správu vláken, paměti a dalších systémových prostředků. Z předchozího plyne, že pro spuštění aplikace musí být v operačním systému nainstalována příslušná verze .NET Frameworku. Další informace ohledně požadovaného hardwaru a softwaru uvádím v příloze A.

3.1.1 Překlad aplikace

Platforma .NET využívá dvoufázový překlad. V první fázi se provádí překlad všech zdrojových kódů do jazyka **MSIL** (*Microsoft Intermediate Language*). Druhá fáze probíhá až během spuštění programu, kdy CLR přeloží MSIL na kód specifický pro daný hardware. Tento způsob překladu umožňuje lépe optimalizovat spouštění aplikace. Za běhu se překládají pouze potřebné části programu (**JIT** - *Just in Time*). Velmi výhodnou vlastností platformy .NET je možnost interoperability kódů napsaných v různých programovacích jazycích.

3.1.2 MSIL

Jedná se o objektový jazyk, který je striktně typový. Není tedy možné používat proměnné bez přesně určeného datového typu. Základní datové typy jsou definovány v CTS (*Common Type System*). V první fázi překladu jsou všechny základní datové typy použité ve zdrojových kódech převedeny na typy z CTS. Rozlišují se hodnotové a referenční datové typy. Hodnotové typy jsou ukládány na zásobník (**Stack**) a proměnné obsahují přímo data. Referenční typy jsou ukládány v paměťové oblasti označované jako spravovaná halda (**Managed Heap**). U referenčního typu obsahuje proměnná pouze adresu, na níž se příslušná data nacházejí.

Pro ošetřování chyb je implementován velmi rozsáhlý systém výjimek. O správu paměti se stará **GC** (*Garbage Collector*), který udržuje informaci o počtu referencí na každý objekt. Pokud žádná reference neexistuje, pak je objekt z paměti odstraněn. K uvolnění však nedochází okamžitě ale v neurčitých časových intervalech, a také pokud docházejí systémové prostředky. Vyčištění paměti lze vyvolat i programově, pokud je třeba odstranit větší množství objektů. Paměť je přidělována na výše zmiňované haldě, kterou si spravuje sama platforma .NET. Odtud vznikl název spravovaný kód.

3.1.3 Jmenné prostory

Pro lepší přehlednost kódu se používají tzv. jmenné prostory (*Namespaces*). Seskupují datové typy (třídy) do logických celků. Tyto celky lze do sebe vnořovat a vytvořit tak stromovou strukturu. V různých jmenných prostorech pak mohou existovat třídy se stejným názvem a rozdílnou funkcí, aniž by docházelo ke kolizi. Při použití takovéto třídy se podle názvu jmenného prostoru rozliší, která verze se má použít při překladu.

Podrobnější informace o platformě .NET a programovacím jazyku C# lze nalézt na internetových stránkách **MSDN** (*Microsoft Developer Network*) [5] nebo v odborné literatuře [6] a [7]. Informace o Windows API naleznete na internetových stránkách [8].

3.2 Přenositelnost na Linux

K překladu a spouštění spravovaného kódu musí být v OS nainstalován .NET Framework. Microsoft však podporuje tuto platformu pouze na Windows. V posledních několika letech byly proto vyvinuty dvě alternativní implementace .NETu pro další operační systémy.

3.2.1 MONO

Jedná se o „open source“ projekt, který je založen na velkém množství technologií vyvinutých a standardizovaných společností Microsoft. Prozatím je podporován pouze C# a Visual Basic (VB). Vývoj tohoto alternativního frameworku finančně podporuje společnost Novell.

Aktuální Mono (1.9) má implementovánu většinu knihoven z .NETu 2.0 a některé části verze 3.0. Pro vývoj grafických rozhraní (**GUI**) jsou k dispozici knihovny Windows Forms nebo GTK#. Knihovny Windows Forms však ještě nejsou zcela dokončené. Knihovny GTK# vznikly ve spolupráci s komunitou GNOME. Podrobnější informace o vývoji projektu lze nalézt přímo na webových stránkách [9].

3.2.2 DotGNU Portable.NET

Tento alternativní framework je stejně jako Mono založen na standardech od Microsoftu. Kompilátor podporuje jazyk C#, ANSI C a pracuje se na podpoře VB. Na rozdíl od projektu Mono je cílem DotGNU dosáhnout stoprocentní kompatibility s implementací .NET Frameworku od Microsoftu.

Pro vytváření GUI aplikací je k dispozici pouze knihovna Windows Forms, ale ani u tohoto Frameworku není ještě úplně dokončená. O veškeré vykreslování na obrazovku se stará základní kreslící vrstva napsaná v jazyce C#, což by mělo zajistit lepší emulaci prostředí Windows. Další informace jsou k dispozici na stránkách projektu [10].

Ani v jedné ze zmiňovaných alternativ .NETu nelze zatím aplikaci přeložit bez dodatečných úprav. Pro překlad a spuštění programu je nutné odstranit nebo nahradit některé části nepodporovaného kódu. Jedná se především o kód automaticky generovaný vývojovým prostředím a některé metody, které ještě nejsou implementovány.

3.3 Jazyk XML

XML je obecný značkovací metajazyk, který byl vyvinut a standardizován konsorciem W3C. Umožňuje snadné vytváření konkrétních značkovacích jazyků pro různé účely. Hlavním cílem XML je především zápis strukturovaných dat do dokumentu v textové podobě. Díky standardizaci nedochází k problémům s kompatibilitou na různých operačních systémech.

XML tvoří součást platformy .NET Framework a vývojového prostředí Visual Studio (dále VS). Veškerá nastavení a konfigurační soubory jsou ukládány v tomto formátu. Ve své aplikaci používám XML pro ukládání nastavení aplikace a informací o datovém záznamu. Komunikační protokol mezi serverem a klientem je také založen na XML. V tomto textu není prostor na podrobný popis, a proto uvádím jen potřebné základy. Další podrobnosti lze nalézt na internetových stránkách konzorcia W3C [11]. Velmi podrobný popis uvádí na svých stránkách také Jiří Kosek [12].

Text XML dokumentu obsahuje značky (tagy) a elementy. Značky musí být vždy párové. Základ elementu tvoří otevírací a uzavírací značka. Element může obsahovat data, atributy a další vnořené elementy nebo je prázdný. Každý validní XML dokument obsahuje vždy jen jeden kořenový element, do kterého jsou vloženy všechny ostatní elementy. Tagy vnořených elementů se nesmí křížit.

Výsledný XML dokument je sestaven z několika částí:

1. **Deklarace** – `<?xml version="1.0"encoding="utf-8"?>`, definuje verzi XML, použité kódování dokumentu a případně i další parametry.

2. **DTD** (*Document Type Definition*), obsahuje deklaraci povolených elementů dokumentu. Blok DTD může být vložen přímo v XML dokumentu nebo v externím souboru. Jeho uvedení není povinné.

3. **Data dokumentu** – data strukturovaná dle předchozích pravidel.

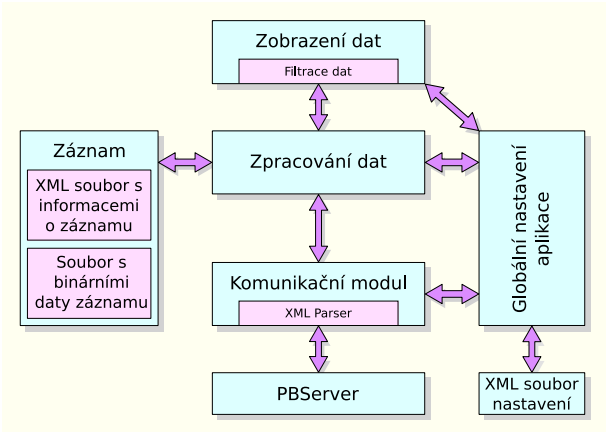
Příklad XML dokumentu bez definice DTD:

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
   <NázevElementu>Data elementu</NázevElementu>
   <PrázdnýTag />
   <ElementSAtributy attr1="hodnota2" attr2="hodnota2" />
   <Vnoření>
      <data>1. vnořený element.</data>
      <data>2. vnořený element.</data>
   </Vnoření>
</root>
```

Pro zpracování XML dokumentu obsahuje platforma .NET namespace `System.XML`. Způsobů pro čtení, ukládání a zpracování XML dokumentu je více. Popis některých z nich provedu vždy v místě jejich použití.

3.4 Struktura aplikace

Klientská aplikace analyzátoru je velmi rozsáhlá, proto jsem ji rozložil na funkční celky uvedené na obrázku 3.2. Popis jednotlivých bloků obsahují následující kapitoly.



Obrázek 3.2: Blokové schéma aplikace

Aplikaci vyvíjím v prostředí Visual Studio 2005. V tomto prostředí se zdrojové kódy ukládají do tzv. **Solution** (řešení). V rámci řešení člením soubory zdrojových kódů do několika projektů a dále do jmenných prostorů (**Namespaces**). Každý projekt obsahuje namespace s názvem projektu. Hlavní projekt aplikace se překládá jako spustitelný (exe) soubor a ostatní projekty jako knihovny (dll).

Názvy a obsah jednotlivých projektů aplikace:

- `Communication` – komunikační modul, řeší veškeré záležitosti, které se týkají spojení mezi serverem a klientem (komunikační protokol a jeho stavový automat).
- `DataProcessing` – obsahuje vše potřebné pro načítání, ukládání a zpracování dat ze serveru nebo datového záznamu.
- `NewControls` – projekt s upravenými komponentami pro grafické rozhraní aplikace.
- `PB_Remote_Analyzer` – hlavní projekt programu, obsahuje formuláře grafického rozhraní a vstupní bod aplikace (Program.cs).
- `Settings` – obsahuje definice tříd sloužících pro správu nastavení aplikace.

Kapitola 4

Nastavení aplikace

Je zcela nevhodné, aby uživatel musel konfigurovat aplikaci při každém spuštění. Proto jsem implementoval systém pro správu nastavení, který popisuji v první části této kapitoly. V druhé části se věnuji problematice jazykové lokalizace programu.

4.1 Správa nastavení

Vývojové prostředí Visual Studio (VS) a jazyk C# jsou velmi flexibilní a umožňují ukládat nastavení mnoha různými způsoby. Zde uvádím některé z nich:

- Ukládání položek do systémového registru.
- Využití systému integrovaného ve VS.
- Vlastní implementace správy nastavení.

První zmiňovaný způsob označuji za nevyhovující. Aplikace neobsahuje odinstalační nástroj, který by se postaral o vyčištění registru při odstranění souborů programu. V registru pak zůstávají nepoužívané záznamy a jeho velikost zbytečně narůstá.

4.1.1 Integrovaná správa nastavení

Ve VS lze u komponent grafického rozhraní určit vlastnosti, jejichž hodnoty mají být ukládány do XML souboru. Pokud soubor s nastavením neexistuje, tak se automaticky vygeneruje v systémovém adresáři **/Application Settings/...** u jednotlivých uživatelů.

Při spuštění programu se samočinně aplikuje nastavení ze souboru a při uzavírání se naopak změny uloží. Tento způsob správy konfigurace jsem používal u prvních verzí aplikace, kde bylo třeba uchovávat pouze vlastnosti některých komponent hlavního okna programu. Později se množina položek nastavení rozrostla i o další prvky, které nesouvisejí s komponentami. Tyto položky však nelze zmiňovaným způsobem uložit, a proto jsem se rozhodl vytvořil vlastní správu nastavení.

4.1.2 Vlastní správa nastavení

Stejně jako v předchozím případě uchovávám konfiguraci aplikace v XML souboru. Uložení a načtení konfigurace provádím pomocí serializace a deserializace instance třídy `GlobalSettings`. Tento postup vyžaduje daleko více kódu než předchozí, ale jeho výhodou je stoprocentní kontrola nad správou nastavení.

Třída, která má být serializována musí splňovat určité podmínky. Do XML souboru se uloží pouze hodnota proměnné nebo vlastnosti (*Property*) s přístupovým modifikátorem[1] `public`. Vlastnosti musí mít implementován blok `get` i `set`, aby bylo umožněno čtení i zápis hodnot nastavení. Metody obsažené ve třídě se v XML souboru nijak neprojevují. Název XML elementu se shoduje s názvem proměnné nebo vlastnosti.

Ve vlastnostech lze použít i referenci na jinou třídu, která obsahuje serializovatelné položky. Tímto způsobem je vytvořeno nastavení se stromovou strukturou. V XML souboru jsou prvky z tříd na nižších úrovních stromu vkládány do elementů, které přísluší třídám na vyšších úrovních. Stromovou strukturu používám i ve své aplikaci. Strom s názvy tříd obsahuje tabulka 4.1. Všechny třídy týkající se nastavení jsou uloženy ve jmenném prostoru `Settings.Sources`.

`GlobalSettings`	– `BasicSettings`	– `SplashScreen`
		– `WindowLayout`
		– `LanguageSettings`
	– `DataSettings`	
	– `ComSettings`	
	– `FilterSettings`	

Tabulka 4.1: Stromová struktura tříd s nastavením

[1] Význam a způsob použití vlastností a přístupových modifikátorů je podrobně vysvětlen v programátorské příručce [6, Kapitola 3]

Metody potřebné pro serializaci a deserializaci XML jsou umístěny ve jmenném prostoru `System.Xml.Serialization`. Načtení nastavení do instance třídy `GlobalSettings` provádím ve statické třídě `Program` při spuštění aplikace. Viz úryvek kódu:

```
GlobalSettings globalSettings = null;
...
   StreamReader sr = null;
   try
   {
      sr = new StreamReader("Cesta k souboru.");
      XmlSerializer xs = new XmlSerializer(typeof(GlobalSettings));
      globalSettings = (GlobalSettings)xs.Deserialize(sr);
   }
```

Při výjimce `FileNotFoundException` vytvářím nový soubor, do kterého ukládám instanci třídy `GlobalSettings` s výchozími hodnotami. Obdobně ukládám nastavení v metodě `saveSettings()`, která je obsažena ve třídě `GlobalSettings`. Stačí v následujícím kódu vynechat řádek `globalSettings = new GlobalSettings();`.

```
...
StreamWriter sw = null;
try
{
   sw = new StreamWriter("Cesta k souboru.", false);
   XmlSerializer xs = new XmlSerializer(typeof(GlobalSettings));
   globalSettings = new GlobalSettings();
   xs.Serialize(sw, globalSettings);
}
...
```

Pro ilustraci přikládám obsah třídy s nastavením úvodní obrazovky aplikace a k ní příslušný blok XML souboru. Proměnná `showSplashScreen` rozhoduje o tom, zda bude obrazovka zobrazena. Proměnná `splashTime` udává čas v sekundách, po který bude okno úvodní obrazovky viditelné. Výchozí hodnoty parametrů jsou nastavovány pomocí konstruktorů, které v příkladu také uvádím.

```
public class SplashScreen
{
   private bool showSplashScreen;
   private byte splashTime;
```

```
   public SplashScreen(bool showSplashScreen, byte splashTime)
   {
      this.showSplashScreen = showSplashScreen;
      this.splashTime = splashTime;
   }

   public SplashScreen() : this(true, 5) { }

   public bool ShowSplashScreen
   {
      get { return showSplashScreen; }
      set { showSplashScreen = value; }
   }

   public byte SplashTime
   {
      get { return splashTime; }
      set { splashTime = value; }
   }
}
```

```
<GlobalSettings>
  <BasicSettings>
    <SplashScreen>
      <ShowSplashScreen>false</ShowSplashScreen>
      <SplashTime>1</SplashTime>
    </SplashScreen>
    ...
```

Kód ostatních tříd je koncipován obdobným způsobem. O jejich obsahu se zmiňuji v následujících kapitolách.

4.2 Jazyková lokalizace aplikace v .NETu

Většina moderních aplikací obsahuje možnost změny tzv. místního a jazykového nastavení. Visual Studio má přímo integrované nástroje pro vytvoření jazykově lokalizované aplikace. Tyto nástroje lze však používat pouze na obsah formulářů. Ostatní dialogy, texty a obrázky, které nejsou obsaženy přímo na formuláři, musí být lokalizovány ručně.

4.2.1 Lokalizace formulářů

Pro aktivaci jazykové lokalizace musí být v okně **Properties** formuláře nastavena vlastnost `Lokalizable` na hodnotu `true`. Vývojové prostředí pak automaticky vytvoří XML soubor prostředků (*Resources*). Do něj vloží všechny textové řetězce, obrázky a vlastnosti udávající velikost a polohu veškerých komponent příslušného formuláře. Kromě toho se změní implementace metody `InitializeComponent()`, která se stará o inicializaci prvků na formuláři při spuštění aplikace. Ve zmíněné metodě VS vytvoří instanci třídy `System.Resources.ResourceManager`, jejíž metoda `GetObject()` je následně používána k načítání prostředků z externího souboru. Hodnoty tedy nejsou uloženy přímo v kódu.

Po aktivaci jazykové lokalizace se vytvoří soubor prostředků pouze pro tzv. neutrální jazyk (*Default*). Resources pro další jazyky vytváří VS automaticky při změně vlastnosti `Language` příslušného formuláře. U každého zvoleného jazyka následně stačí přímo v návrháři VS změnit texty, obrázky a upravit rozmístění komponent. Pokud některý z prostředků není změněn, pak se použije hodnota z neutrální jazykové verze.

Lokalizaci formulářů lze provést i bez instalace vývojového prostředí. .NET Framework SDK (*System Developer Kit*) pro tento účel obsahuje lokalizační nástroj *Windows Resource Localization Editor*.

Výše zmíněná metoda `GetObject()` volí příslušný soubor prostředků podle kódu jazykové verze a oblasti zadaného ve vlastnosti `CurrentUICulture` aktuálního vlákna. Pro český jazyk je tento kód `cs-CZ`. Například u hlavního formuláře aplikace s názvem `MainForm` se pro jazyk default vytvoří soubor prostředků `MainForm.resx`. Pro anglický jazyk má soubor s prostředky název `MainForm.en-GB.resx`.

4.2.2 Lokalizace dialogů

Lokalizace dialogů a ostatních textů je o něco složitější. Stejně jako u formulářů musí být vytvořeny soubory prostředků pro jednotlivé jazyky. Není třeba psát přímo XML kód, protože VS obsahuje nástroj pro tvorbu prostředků. V podstatě stačí v **Solution Manageru** vytvořit nový **Resources File** a pomocí vestavěného editoru do levého sloupce tabulky zapsat název a do pravého textový řetězec. Pro pojmenování souborů platí stejná pravidla, která jsem zmínil v předchozím odstavci. Pomocí editoru lze vkládat i obrázky a další typy prostředků.

Načítání prostředků ze souborů provádím i v tomto případě pomocí instance třídy `System.Resources.ResourceManager`, kterou inicializuji příkazem:

```
ResourceManager resMan = new ResourceManager(
   "PB_Remote_Analyzer.Resources.MainLocales", Assembly.GetExecutingAssembly());
```

První argument udává název jmenného prostoru a souboru s prostředky (bez přípony a kódu jazykové verze). Druhý argument obsahuje referenci na tzv. *sestavení*, ve kterém jsou uloženy informace o souborech s prostředky.

`ResourceManager` má implementováno několik metod pro načítání prostředků. Například pro získání určitého textového řetězce používám příkaz `resMan.GetString(název)`. Argument *název* obsahuje jméno textového řetězce ze souboru prostředků.

Aktuálně zvolený jazyk programu a seznam podporovaných jazyků uchovávám v globálním nastavení aplikace. Pro tyto účely slouží třída `LanguageSettings`, jejíž instance je vnořena do třídy `GlobalSettings` (viz tabulka 4.1). Nastavení jazyku aplikace provádím v konstruktoru hlavního formuláře (`MainForm`):

```
string culture = globalSettings.GeneralSettings.LanguageSettings.CurrentLanguage;

Thread.CurrentThread.CurrentCulture = new CultureInfo(culture);
Thread.CurrentThread.CurrentUICulture = new CultureInfo(culture);
```

Poslední dva řádky zajistí, že celá aplikace bude používat národní prostředí a jazyk uvedený v proměnné `culture`. Uživatel může vybrat jazyk na formuláři nastavení, o kterém se zmíním v další kapitole. Změna místního a jazykového nastavení se projeví vždy až po restartu programu.

Kapitola 5

Grafické rozhraní aplikace

Vývoj grafického uživatelského rozhraní (**GUI**) zabral nemalé množství času, který jsem věnoval celému programu. V podkapitole 3.4 jsem uvedl základní členění aplikace na projekty. Formuláře (okna) GUI jsou součástí projektu `PB_Remote_Analyzer`. Na formulářích používám také upravené a nově vytvořené komponenty ze jmenného prostoru `NewControls.MyControls`. O těchto komponentách se zmiňuji v části této kapitoly.

Složky jmenného prostoru `PB_Remote_Analyzer`:

- `Forms` – složka se zdrojovými kódy formulářů.
- `Images` – zde jsou uloženy všechny potřebné ikony a obrázky.
- `Resources` – obsahuje soubory prostředků pro jazykovou lokalizaci GUI.
- `Sources` – obsahuje pouze vstupní bod aplikace (Program.cs).

Jednotlivé formuláře aplikace jsem se snažil vytvořit s ohledem na základní požadavky pro návrh GUI. Důraz jsem kladl především na přehlednost a jednoduché ovládání. Data a většinu informací zobrazuji uvnitř hlavního okna aplikace. Pro konfiguraci programu a vzdáleného serveru je určeno několik dalších oken. Názvy a funkci všech tříd s formuláři, které jsou obsaženy ve jmenném prostoru `PB_Remote_Analyzer.Forms`, uvádím v následujícím seznamu:

- `ComConfigForm` – formulář určený pro konfiguraci spojení a vzdáleného serveru. Obsahuje nastavení filtrace a podmíněného spouštění záznamu dat (*Triggeru*).
- `FilterForm` – formulář pro nastavení filtru zobrazených dat.
- `MainForm` – hlavní okno aplikace.

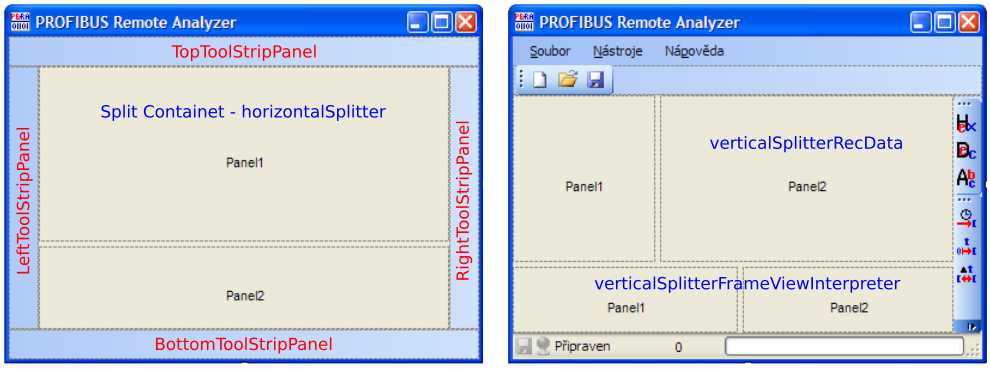
- `SettingsForm` – formulář pro konfiguraci parametrů programu.
- `SplashForm` – úvodní obrazovka programu. Pokud je aktivována v nastavení, tak se zobrazí během inicializace hlavního okna aplikace.
- `StatisticsForm` – formulář pro zobrazení statistik vypočítaných z načtených dat.

V této kapitole popisuji pouze složení okna `MainForm` a `SettingsForm`. Popis ostatních formulářů obsahují další kapitoly. Na formulářích používám komponenty ze jmenného prostoru `System.Windows.Forms`. Informace o vlastnostech a možnostech jednotlivých komponent jsem čerpal z knihy [13].

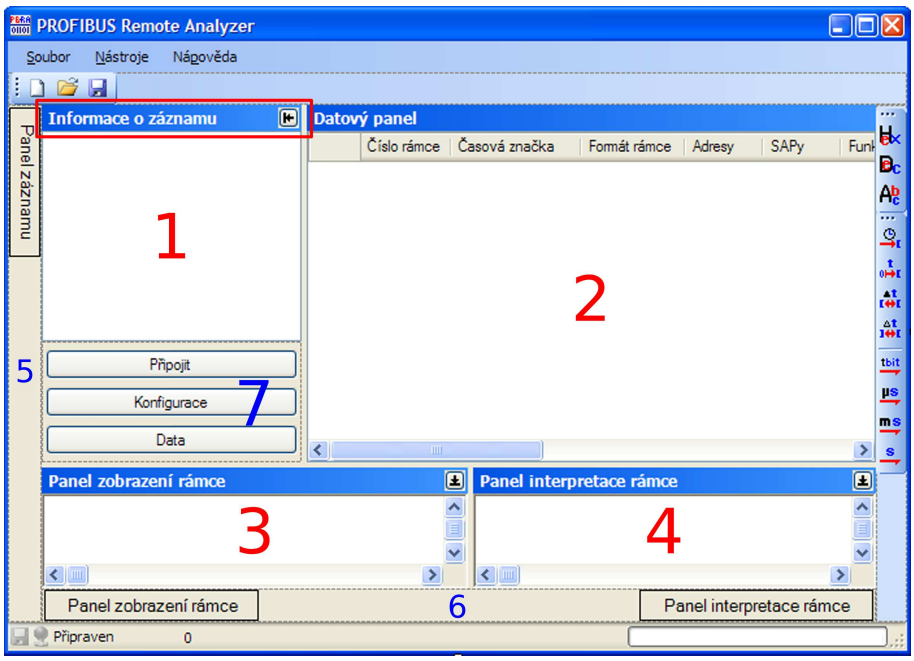
5.1 Struktura hlavního okna aplikace

Hlavní okno aplikace, uložené ve třídě `MainForm`, jsem vytvořil z několika vrstev. Na obrázku 5.1 vlevo je zobrazena první vrstva, kterou tvoří `ToolStripContainer`. Tato komponenta rozděluje formulář na několik částí. Červeně popsané panely jsou určeny pro vložení prvků typu `ToolStrip`, `MenuStrip` nebo `StatusStrip`. Prázdné panely jsou za běhu aplikace skryté. Pomocí myši lze mezi panely přetahovat jednotlivé prvky, pokud nejsou blokované.

Prostřední část `ToolStripContaineru` vyplňuje komponenta typu `SplitContainer`. Tento prvek rozděluje vnitřek formuláře na dvě části, jejichž poměr je možné za běhu aplikace měnit pomocí posuvného oddělovače. Do každé části jsem vložil ještě jeden `SplitContainer`, a tím vznikly čtyři panely. Výsledkem je struktura zobrazená na obrázku 5.1 vpravo, kde jsem také doplnil několik nástrojových lišt, menu a stavový panel.



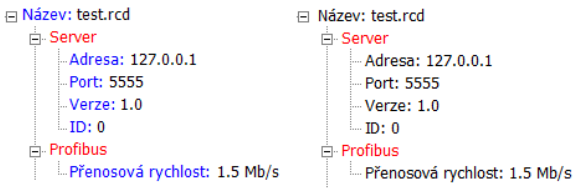
Obrázek 5.1: Základní struktura hlavního okna aplikace



Obrázek 5.2: Výsledné okno aplikace

Na obrázku 5.2 uvádím kompletní formulář z návrháře vývojového prostředí. Panel s číslem 1 používám pro zobrazení informací o datovém záznamu a spojení se serverem. Pro tento účel jsem zvolil komponentu typu `TreeView` (`treeViewRecord`). V závorce uvádím název instance použité v programu. Zmiňovaný prvek zobrazuje položky v přehledném stromu, ale při větším počtu položek informace splývají.

`TreeView` umožňuje ve výchozím režimu pouze změnu barvy u celého řádku. Proto jsem nastavil vlastnost `DrawMode` na hodnotu `OwnerDrawText`, a tím jsem aktivoval uživatelský režim vykreslování textu. Nový kód pro kreslení položek jsem implementoval v metodě `treeViewRecord_DrawNode(...)`, kterou volám při události `DrawNode`. Rozdíl mezi uživatelským způsobem vykreslování textu a výchozím je patrný z následujícího obrázku. Nový režim zobrazení položek je podle mého názoru daleko přehlednější.



Obrázek 5.3: Porovnání režimů vykreslování položek u `TreeView`

Úkolem panelu s číslem 2 je zobrazení dat v přehledné tabulce. Pro tyto účely .NET Framework obsahuje několik komponent. Nejvhodnější je `DataGridView` (`dataTable`). Podrobnosti o způsobu načítání dat do tabulky uvádím až v kapitole 7.4.

Panely s číslem 3 a 4 slouží k zobrazení doplňkových informací o rámci, který je vybrán v tabulce s daty. Panel vlevo zobrazuje hexadecimální výpis obsahu rámce a panel vpravo interpretaci tohoto rámce podle protokolu Profibus DP. V obou panelech je vložena komponenta typu `RichTextBox`, do které lze načíst text bez formátování nebo s formátováním dle specifikace **RTF**.

Dosud jsem nezmínil jednu významnou funkci `SplitContainerů`, které používám na pozadí formuláře. Nastavením vlastnosti `Panel1Collapsed` nebo `Panel2Collapsed` na hodnotu `true` se příslušný panel skryje. Ve své aplikaci umožňuji skrývání panelů s čísly 1, 3 a 4. Po schování všech tří zmiňovaných panelů se tabulka s daty roztáhne na většinu plochy formuláře. Ke skrytí panelu dojde po stisku tlačítka u komponenty typu `MyTitleBar`. Na obrázku 5.2 jsem jeden z těchto prvků označil červeným obdélníkem. U panelu s datovou tabulkou je jako titulek použita komponenta `MyLabel`, která neobsahuje skrývací tlačítko.

Části formuláře, které jsou označené čísly 4 a 5, slouží pro opětovné zobrazení skrytých panelů. Tlačítka a lišty jsou viditelné pouze v případě, že je příslušný panel skrytý. Po stisku tlačítka se panel opět zobrazí a lišta zmizí. Tlačítko (`Button`) ze jmenného prostoru `System.Windows.Forms` bylo pro tyto účely nevyhovující, protože neumí zobrazovat text svisle. To mě vedlo k vytvoření vlastní komponenty `MyButton`, u které jsem tuto vlastnost implementoval.

Poslední nepopsanou částí formuláře je panel s číslem 7, který obsahuje tři tlačítka. Tento panel je viditelný pouze při vytváření nového datového záznamu. První tlačítko slouží pro navázání/ukončení spojení se serverem. Po stisku druhého tlačítka se zobrazí formulář `ComSettingsForm`, na kterém lze nakonfigurovat spojení a další parametry serveru. Poslední tlačítko spouští/ukončuje čtení dat ze serveru.

U panelů, které jsou na obrázku 5.2 označeny čísly 1 – 4, může uživatel měnit také rozmístění v rámci okna. Rozmístění však nemůže být zcela libovolné. V aplikaci implementuji pouze záměnu obsahu panelů uvnitř jednoho `SplitContaineru`. Lze tedy zaměnit horní a dolní dvojici panelů a následně panely uvnitř dvojic. Podle mého názoru těchto několik možností zcela vyhovuje požadavkům na maximální pohodlí uživatele. Konfigurace rozmístění panelů je přístupná na formuláři nastavení `FormSettings`, který popisuji v závěru této kapitoly.

5.2 Upravené a nové komponenty

Každý formulář a všechny jeho prvky, jsou objekty popsané třídami. Objektově orientované programování umožňuje dědění tříd. Zděděná třída (potomek) přebírá kód předka, který lze dále rozšířit. Jazyk C# podporuje i tzv. *vizuální dědičnost*. Tento způsob dědění je určen pro komponenty GUI, kde potomek zdědí od předka i vizuální podobu. Zmiňovanou technikou lze velice efektivně implementovat nové prvky GUI bez nutnosti jejich návrhu od samého základu. V následujícím textu popisuji upravené komponenty ze jmenného prostoru `NewControls.MyControls`.

5.2.1 Komponenta MyLabel

`MyLabel` jsem vytvořil speciálně pro titulek panelu s číslem 2 zobrazeného na obrázku 5.2). Komponenta je vizuálně zděděna od třídy `System.Windows.Forms.Label` a má přetíženou metodu `OnPain(...)`. V této metodě implementuji kód pro vykreslení barevného lineárního gradientu na pozadí textu. Zde uvádím část zdrojového kódu:

```
protected override void OnPaint(PaintEventArgs e)
{
   Graphics gfx = e.Graphics;
   Rectangle rectLabel = new Rectangle( ... )
   StringFormat strFormat = new StringFormat();
   ...

   Brush captionBrush = new LinearGradientBrush(ClientRectangle,
      SystemColors.ActiveCaption, SystemColors.GradientActiveCaption,
      0.0);
   Brush textBrush = new SolidBrush(this.ForeColor);

   gfx.FillRectangle(captionBrush, ClientRectangle);
   gfx.DrawString(this.Text, this.Font, textBrush, rectLabel, strFormat);
}
```

Nejprve inicializuji instanci bázové třídy `Graphics`, která v sobě zapouzdřuje tzv. *kontext zařízení*. Tento kontext se stará o vykreslovaní grafiky na výstupní zařízení[1]. Na dalších řádcích je provedena inicializace kreslící plochy, formátu textu a typů štětců. Na posledních dvou řádcích volám metody bázové třídy `Graphics`, které provedou vykreslení pozadí s lineárním gradientem a textu.

[1]Podrobný popis kreslícího rozhraní GDI+ je obsažen v odborné literatuře [6, kapitola 19].

5.2.2 Komponenta MyTitleBar

Vizuální podoba komponenty je zobrazena již na obrázku 5.2, kde je zvýrazněna červeným obdélníkem. `MyTitleBar` jsem složil z prvku `MyLabel` a obyčejného čtvercového tlačítka typu `System.Windows.Forms.Button`. Při stisku tlačítka je generována událost, kterou jsem definoval následujícím způsobem:

```
public delegate void delegateCollapseButton_Clicked(object Sender, EventArgs e);
public event delegateCollapseButton_Clicked CollapseButton_Clicked;
```

V prvním řádku uvádím *delegát*, který slouží jako reference na metodu ošetřující událost. Metoda musí mít stejnou návratovou hodnotu a argumenty jako delegát. Druhý řádek obsahuje definici samotné události. Pro vyvolání této události stačí použít příkaz `CollapseButton_Clicked(this, new EventArgs())`. Následně se provede metoda, na níž ukazuje delegát. Další podrobnosti o použití delegátů a událostí jsou obsaženy v programátorské příručce [6, kapitola 4].

5.2.3 Komponenta MyButton

Třída `System.Windows.Forms.Button` neumožňuje vykreslení svislého textu. Proto jsem od ní vizuálně zdědil komponentu `MyButton`, kde jsem tuto možnost doplnil. Vlastnost `Text` rodičovské třídy jsem nahradil vlastností s názvem `ButtonText`. Dále jsem přidal novou vlastnost `DirectionOfText`, ve které může být podle požadovaného směru textu nastavena hodnota `Vertical` nebo `Horizontal`. Podobně jako u komponenty `MyLabel` vykresluji text v přetížené metodě `OnPain(...)`.

```
protected override void OnPaint(PaintEventArgs pevent)
{
   base.OnPaint(pevent);

   RectangleF clipRectangle = pevent.ClipRectangle;
   SolidBrush solidBrush = new SolidBrush(this.ForeColor);
   StringFormat stringFormat = new StringFormat();

   if (directionOfText == TextDirection.Vertical)
   {
      ...
      stringFormat.FormatFlags = StringFormatFlags.DirectionVertical;
   }
```

```
    ...
    pevent.Graphics.DrawString(this.buttonText, this.Font, solidBrush,
       clipRectangle, stringFormat);
}
```

Na začátku volám metodu `OnPaint(...)` z rodičovské třídy `Button`, aby se nakreslilo samotné tlačítko. Dále inicializuji štětec a instanci třídy `StringFormat`. Pomocí této třídy je možné nastavit různé parametry textu. Například směr textu lze jednoduše změnit přiřazením flagu `DirectionVertical` (viz úryvek zdrojového kódu). Nakonec volám metodu `DrawString(...)`, která vykresluje text dle parametrů zadaných jako argumenty.

5.3 Položky nastavení

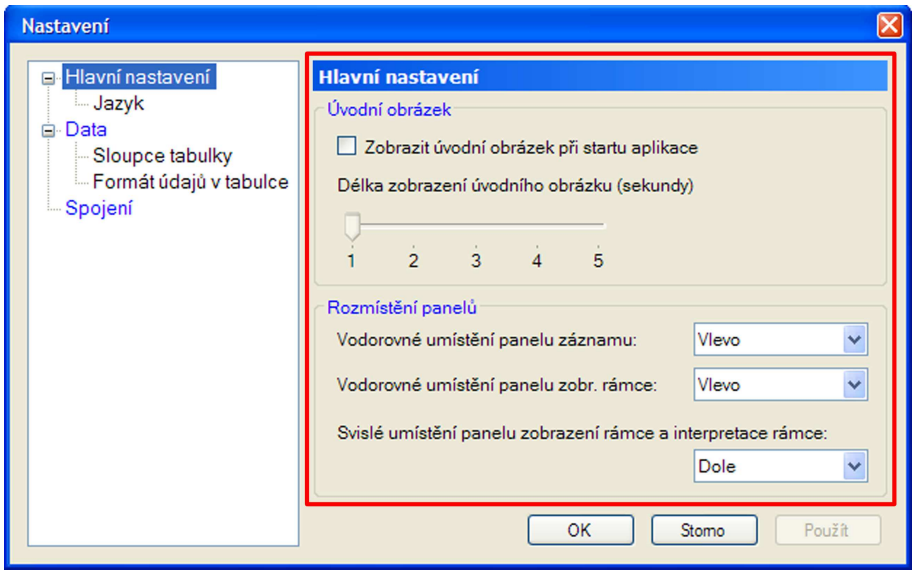
Způsob správy nastavení aplikace jsem popsal v kapitole 4. Pro ukládání položek, které se týkají hlavního okna aplikace, je určena třída `WindowLayout`. Její umístění uvnitř hlavní třídy nastavení `GlobalSettings` ukazuje tabulka 4.1.

Instance třídy `WindowLayout` obsahuje velikost hlavního okna aplikace, rozmístění panelů a polohu posuvných oddělovačů mezi nimi. Dále u každého panelu uchovávám informaci o tom, zda je viditelný nebo skrytý.

Ukládání nastavení aplikace provádím v metodě `mainForm_FormClosing(...)`, která je volána při události `FormClosing`. Po nastavení všech potřebných parametrů v instanci třídy `WindowLayout` se provede serializace celé konfigurace programu do XML souboru. Postup serializace jsem popsal v podkapitole 4.1.2. Při spuštění programu provádím nastavení komponent v konstruktoru formuláře `MainForm`.

5.4 Formulář nastavení

Formulář s nastavením programu (`SettingsForm`) lze zobrazit pomocí položky menu v hlavním okně aplikace (**Nástroje** → **Nastavení aplikace**). Uspořádání komponent na formuláři nastavení ukazuje obrázek 5.4. V levé části okna je umístěna komponenta typu `TreeView` (`treeViewSettings`). V pravé části jsem červeným obdélníkem vyznačil prostor, který vyplňuje komponenta `TabControl` (`tabControlSettings`). Formulář je dále doplněn trojicí klasických ovládacích tlačítek.



Obrázek 5.4: Formulář s nastavením aplikace

Komponenta `tabControlSettings` obsahuje stránky s konfigurací aplikace. Vždy je viditelná pouze jedna ze stránek. V běžném případě jsou na vrchní straně `TabControlu` zobrazeny záložky, pomocí kterých lze přepínat viditelnou stránku. V okně nastavení jsem však tyto záložky zmenšil na minimální možnou velikost a překryl titulkem, aby nebyly viditelné.

K přepínání mezi jednotlivými částmi nastavení používám místo záložek komponentu `treeViewSettings`, kde jsou názvy všech stránek uspořádány do stromové struktury podle obsahu. Stačí kliknout myší na určitou položku v `treeViewSettings` a následně se přenastaví `tabControlSettings`.

Tento způsob přepínání jsem zvolil hlavně kvůli přehlednosti. Při velkém počtu stránek v komponentě `TabControl` je nutné posouvat lištu se záložkami, a tím se snižuje přehlednost. Druhou možností je nastavit víceřádkový režim záložek, pak ale zabírají větší prostor.

Položky nastavení na jednotlivých stránkách jsou inicializovány v konstruktoru formuláře `SettingsForm`, kterému jako argument předávám instanci třídy `GlobalSettings`. Při změně některé z položek se u příslušné stránky nastaví vlastnost s názvem `Tag` na hodnotu `true`. Následně se po stisku tlačítka **Ok** nebo **Použít** zavolá metoda `applySettings()`, která uloží a provede změny v nastavení.

Kapitola 6

Komunikace

Komunikace tvoří jednu z klíčových částí této práce. Klientská a serverová aplikace analyzátoru jsou spouštěny jako dva nezávislé procesy, které spolu musí komunikovat. Nejvhodnějším řešením pro tento problém je použití tzv. **soketu**.

Původní soketové rozhraní bylo vyvinuto na univerzitě v **Berkeley**. V operačních systémech Windows je integrováno rozhraní **Windows Socket API** kompatibilní s původní verzí. Platforma .NET Framework obsahuje vše potřebné pro práci se sokety ve jmenném prostoru `System.Net.Sockets`. Pomocí soketového rozhraní se mezi procesy vytváří virtuální komunikační kanál na úrovni transportní vrstvy ISO/OSI modelu. Lze tedy předávat data mezi procesy spuštěnými na různých počítačích, jež mohou mít i rozdílné operační systémy. Nutná je pouze existence vhodného fyzického propojení počítačů (např. síť typu Ethernet) a kompatibilní implementace soketového rozhraní v OS.

Z maximální přenosové rychlosti sběrnice Profibus (12 Mb/s) vyplývá požadavek pro přenos poměrně velkého objemu dat mezi serverovou a klientskou aplikací. Pro tyto účely jsem jako komunikační médium zvolil síť typu Ethernet s přenosovou rychlostí minimálně 100 Mb/s. Pro online přenos dat je vhodné, aby klient i server byli připojeni ve stejné síti LAN. Internetové spojení lze využít pouze pro offline přenos dat, která již budou uložena na serveru. Tato funkce však zatím není implementována.

U soketu jsem jako transportní protokol zvolil **TCP** (*Transmision Control Protocol*) z „rodiny“ protokolů TCP/IP. Ze všech možných protokolů je TCP nejvhodnější, protože je orientovaný na spojení. Před začátkem komunikace se vytvoří mezi klientem a serverem dvoubodové spojení, které je aktivní po celou dobu komunikace. Obě strany spoje jsou identifikovány IP adresou a portem. Vždy je zaručeno doručení paketů ve stejném pořadí, jako byly vyslány. Při chybě je přenos automaticky opakován. Další podrobnosti obsahuje průvodce TCP sítěmi [14].

Data jsou přenášena komunikačním kanálem jako posloupnost bajtů. Formát a význam přijatých zpráv definuje komunikační protokol na úrovni aplikační vrstvy ISO/OSI modelu. Pro tyto účely lze použít některý z již definovaných protokolů nebo implementovat vlastní. V případě analyzátoru používám protokol založený na XML.

V této kapitole postupně popisuji: navázání spojení, vlastní komunikační protokol a způsob přenosu binárních dat ze serveru do klientské aplikace. Pro zdrojové kódy, které se týkají komunikace, jsem vytvořil projekt `Communication` s následujícím obsahem:

- **Resources** – obsahuje soubor prostředků pro jazykovou lokalizaci.
- **Sources** – zde jsou uloženy zdrojové kódy komunikačního modulu.
  - `ComModule` – hlavní třída, která obsahuje zdrojové kódy pro obsluhu spojení a stavový automat komunikace.
  - `ComStates` – zde je uchováván aktuální stav komunikace a definice všech zpráv komunikačního protokolu.
  - `Element` – instance této třídy je používána pro předávání dat mezi jednotlivými částmi komunikačního modulu.

Před popisem komunikace musím nejdříve objasnit způsob uchovávání dat ze serveru a jim příslušných doplňkových informací.

6.1 Popis datového záznamu

V aplikaci umožňuji ukládání dat nasnímaných ze serveru. Data a další potřebné informace o spojení a nastavení serveru sdružuji do tzv. *datového záznamu* (dále jen záznamu). Záznam je v paměti reprezentován jako instance třídy `Record` ze jmenného prostoru `DataProcessing.Records`. Pro lepší přehlednost třída `Record` obsahuje instance dalších tříd, jejichž názvy a význam uvádím v následujícím seznamu:

`PB_Data` – zde jsou uchovávány parametry, které se týkají dat načtených ze serveru: počet rámců; datum a čas počátku sběru dat; přenosová rychlost sběrnice Profibus; název souboru s binárními daty; reference na data přečtená ze serveru ....

`PB_Server` – zde jsou k dispozici informace o IP adrese a portu serveru, ze kterého byla data přečtena. Dále je ve třídě uložena informace o verzi komunikačního protokolu.

`PB_Filter` – obsahuje nastavení datového filtru na straně serveru.

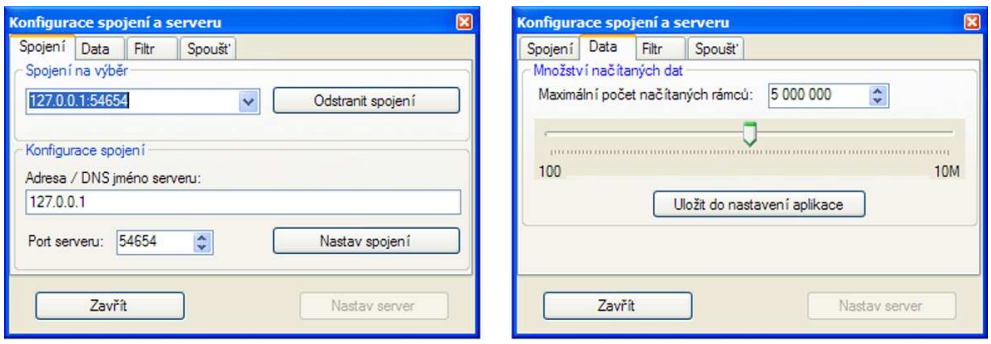
`PB_Trigger` – obsahuje nastavení podmíněného spouštění záznamu dat na straně serveru.

Filtraci a podmíněné spouštění záznamu dat popisuji v podkapitole 7.5. Informace o záznamu jsou zobrazovány v panelu uvnitř hlavního okna aplikace, jehož strukturu jsem podrobně rozebral v kapitole 5. O naplnění komponenty typu `TreeView` se stará metoda `showRecInfo()` obsažená ve třídě `Record`.

V současné verzi programu je záznam ukládán do dvou souborů. XML soubor s příponou `.rcd` obsahuje pouze informace o záznamu. Ve stejnojmenném binárním souboru s příponou `.bin` jsou uloženy jednotlivé rámce načtené ze serveru. Formát ukládaných rámců popisuji v sekci 6.5.1. XML soubor ukládám pomocí serializace instance třídy `Record`. Postup serializace a pravidla pro formát serializovaných tříd jsem uvedl již u správy hlavního nastavení (sekce 4.1.2). Způsob, jakým zpracovávám a uchovávám data v paměti, podrobněji popisuji v kapitole 7.

6.2 Vytvoření komunikačního kanálu

Komunikace se serverem je umožněna pouze při založení nového záznamu. Stačí v menu na hlavním formuláři aplikace zvolit položku **Soubor** → **Nový**. Po tomto kroku jsou inicializovány nové instance tříd `Record`, `ComModule` a `DataReader`. Třída `DataReader` obsahuje zdrojové kódy pro zpracování dat z binárního souboru a ze serveru. Je uložena ve jmenném prostoru `DataProcessing.Records`. V panelu s informacemi o záznamu se objeví panel se třemi tlačítky, který je na obrázku 5.2 označen číslem 7. Po stisku tlačítka s nápisem **Konfigurace** se zobrazí formulář `ComConfigForm`.



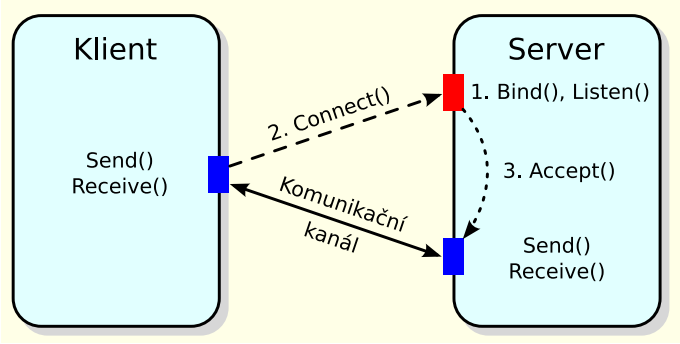
Obrázek 6.1: Formulář `ComConfigForm` pro nastavení spojení a serveru

Na pozadí formuláře `ComConfigForm` je vložena komponenta typu `TabControl`, která obsahuje stránky s konfigurací spojení a serveru. Nastavení spojení je zobrazeno na prvním z obrázků 6.1. Potřebné údaje ukládám do hlavního nastavení aplikace, aby je uživatel nemusel při každém spojení znovu zadávat. Lze tedy vybrat některé z již předdefinovaných nastavení nebo zadat novou IP adresu a port serveru.

Pro ukládání parametrů spojení je vyhrazena třída `ComSettings`, kde je jako datový člen vloženo pole s instancemi třídy `ServerConnections`. Umístění třídy `ComSettings` uvnitř třídy `GlobalSettings` zobrazuje tabulka 4.1. Přidávání a odebírání předdefinovaných spojení jsem umožnil také na jedné ze záložek formuláře `SettingsForm`.

V druhé části obrázku 6.1 je zobrazeno nastavení maximálního počtu rámců načítaných ze serveru. Podrobné informace o zpracování dat a jejich udržování v paměti jsem umístil do kapitoly 7.

Pokus o navázání spojení se odstartuje stiskem tlačítka s nápisem **Připojit**, které je umístěno v dolní části panelu s informacemi o záznamu (viz obrázek 5.2). Po stisku tlačítka se provede kontrola připojovacích údajů vzdáleného serveru. Při správném nastavení je zavolána metoda `socComXML.connect(record.PB_Server.ServerAddress, record.PB_Server.ServerPort)`. Objekt `socComXML` je instancí třídy `ComModule`. Jako argumenty metody `connect()` jsou použity připojovací údaje serveru nastavené v záznamu.



Obrázek 6.2: Schéma navázání spojení mezi klientem a serverem

Obrázek 6.2 ilustruje proceduru vytvoření komunikačního kanálu. V první fázi musí být na obou stranách vytvořen příslušný soket se stejnými parametry transportního protokolu. Na straně serveru je dále pomocí metody `Bind()` k soketu přiřazena lokální IP adresa a port. Následně je metodou `Listen()` aktivován poslech na daném soketu. Tím je server připraven na připojení klienta.

Na straně klienta je zavolána metoda `Connect()`, které je předána IP adresa a port vzdáleného serveru. Pokud server přijme spojení metodou `Accept()`, tak se inicializuje nový soket a vytvoří se komunikační kanál. Na původním soketu může server čekat na připojení dalšího klienta. Pro odesílání a příjem dat jsou na obou stranách dostupné standardní metody `Send()` a `Receive()`.

Nyní popíšu implementaci navázání spojení na straně klientské aplikace. Zdrojový kód je obsažen ve výše zmiňované metodě `ComModule.connect(...)`.

```
clientSocket = new Socket(
      AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
```

V první fázi je vytvořen nový soket. Argument `AddressFamily` definuje způsob adresování jednotlivých soketů. U protokolu TCP je každý soket identifikován pomocí IP adresy a portu. Druhý argument určuje, jakým způsobem jsou předávána data mezi sokety. U TCP protokolu musí být nastavena hodnota `Stream`, protože jsou data předávána jako datový proud paketů (rámců). Každý paket je očíslován a lze tedy rozpoznat výpadek komunikace a vyžádat opakovaný přenos chybějící části. Poslední argument soketu přímo definuje typ transportního protokolu.

```
IPHostEntry hostEntry = Dns.GetHostEntry(host);
IPEndPoint ipEndPoint = new IPEndPoint(hostEntry.AddressList[0], port);
```

Předchozí blok kódu vytvoří koncový bod spojení se zadanými připojovacími údaji. Proměnná `port` obsahuje číslo portu u soketu na straně serveru. Proměnná `host` může obsahovat IP adresu nebo jméno síťové stanice (v tomto případě serveru). Do proměnné `hostEntry` se ukládají informace získané z DNS serveru. DNS (*Domain Name Server*) se stará o překlad IP adres na jména stanic v síti a naopak. Pokud DNS server neobsahuje žádný záznam pro zadanou adresu nebo jméno stanice, tak je vyvolána výjimka `SocketException`.

```
clientSocket.Connect(ipEndPoint);
nsRead = new NetworkStream(clientSocket);
```

Uvedenými příkazy se klient pokusí připojit k serveru, jehož adresa a port jsou metodě `Connect()` předány uvnitř argumentu `ipEndPoint`. Při neúspěšném navázání spojení je opět generována výjimka typu `SocketException`. V opačném případě se vytvoří nový komunikační kanál a instance třídy `NetworkStream`, pomocí níž jsou přijímána data ze soketu.

Nyní je umožněna vzájemná komunikace mezi serverem a klientem. Není však vhodné, aby se o příjem dat ze soketu staralo přímo hlavní vlákno aplikace. Proto pomocí následujících příkazů vytvářím nové pracovní vlákno. Schéma vícevláknového modelu uvádím dále na obrázku 6.3.

```
thrRec = new Thread(new ThreadStart(receive));
thrRec.Start();
```

Příkaz v prvním řádku vytváří nové vlákno a přiřazuje mu metodu `receive()`. Druhým příkazem se vlákno spustí a začne vykonávat kód přiřazené metody. Obsah metody `receive()` popisuji dále v sekci 6.4.

Pokud je v popisované metodě `connect()` generována nějaká výjimka, tak je zachycena a uživateli se zobrazí dialogové okno s popisem chyby. Pokus o navázání spojení se serverem se samozřejmě korektně ukončí.

6.3 Komunikační protokol

V sekci 3.3 jsem uvedl základy značkovacího jazyka XML. V následující části kapitoly popisuji použití XML pro předávání zpráv mezi klientskou a serverovou aplikací. Jednotlivé zprávy jsou reprezentovány jako XML elementy. Pro každý typ zprávy je definován určitý název elementu a jeho obsah (viz tabulka 6.1). Záznam komunikace by tedy vypadal jako obyčejný XML dokument. V podstatě se vytvářejí dva dokumenty, jelikož komunikace je obousměrná.

Syntaxe komunikačního protokolu je dána pravidly pro zápis XML a dvěma DTD soubory, které obsahují definice elementů. Definiční soubory jsou uloženy na přiloženém CD a jejich výpis uvádím také v příloze B.

Po navázání TCP spojení může začít komunikace na úrovni aplikačního protokolu. Na obou stranách musí být jako první odeslána deklarační hlavička XML a otevírací značka kořenového elementu. Stejně jako u běžného XML dokumentu musí být i v tomto případě použit pouze jeden kořenový element. Komunikace končí odesláním uzavírací značky kořenového elementu.

V následující tabulce 6.1 uvádím názvy a význam jednotlivých elementů komunikačního protokolu. Ve druhém sloupci je uvedena strana, která element odesílá (K = klient, S = server).

Název elementu	Strana	Význam
`pbanalyzer`	K, S	Kořenový element dokumentu komunikace
`pbclient`	K	Identifikační zpráva klienta
`pbserver`	S	Identifikační zpráva serveru
`pbconfig`	K, S	Element pro ohraničení konfiguračního bloku
`pbset`	K	Zpráva pro nastavení parametrů na serveru
`pbget`	K	Zpráva pro čtení nastavení parametrů ze serveru
`pbans`	S	Odpověď na zprávy pbset a pbget
`pbdata`	K, S	Element pro ohraničení bloku přenosu dat
`d`	S	Zpráva pro posílání dat ze serveru
`pbdatastop`	K	Požadavek na ukončení přenosu dat

Tabulka 6.1: Definice elementů komunikačního protokolu a jejich význam

6.3.1 Sémantika komunikačního protokolu

Sémantika komunikačního protokolu definuje pořadí zpráv během komunikace. Klientská aplikace automaticky ukončuje spojení při špatném pořadí zpráv nebo příjmu neplatné zprávy. Komunikaci mezi klientskou a serverovou aplikací jsem rozdělil do několika bloků:

1. Zahájení komunikace a identifikace stanic
2. Konfigurační blok
3. Blok přenosu dat
4. Ukončení spojení

Při bezchybné komunikaci jsou jednotlivé bloky protokolu vykonány v uvedeném pořadí. Protokol navíc umožňuje během jedné relace opakovaně přecházet mezi konfiguračním blokem a blokem přenosu dat. Nyní popíšu obsah a pořadí zpráv jednotlivých bloků.

Zahájení komunikace a identifikace stanic

Veškerá komunikace je iniciována ze strany klientské aplikace. Po navázání TCP spojení je automaticky odeslána následující deklarace XML dokumentu a otevírací značka kořenového elementu. Server jako odpověď pošle stejné elementy.

```
<?xml version="1.0" encoding="utf-8"?>
<pbanalyzer>
```

Deklarační hlavička obsahuje atributy se specifikací verze XML a kódování použitého v dokumentu. Po předání těchto zpráv klient odešle svou identifikační zprávu a jako odpověď obdrží identifikaci serveru.

Identifikační zpráva klienta: `<pbclient ver="1.0" id="1"/>`
Identifikační zpráva serveru: `<pbserver ver="1.0" id="0"/>`

Atribut `ver` uvnitř elementů udává verzi komunikačního protokolu. Na straně serveru i klienta musí být stejná verze. V opačném případě je odeslána ukončovací značka kořenového elementu a následně je uzavřen soket. Atribut `id` musí být naopak u každé aplikace unikátní. Pro server je vyhrazena hodnota 0 a klientské aplikace musí mít celočíselný identifikátor v rozsahu 1 – 10.

Po úspěšné výměně identifikačních zpráv klient odešle otevírací element konfiguračního bloku `<pbconfig>`. Server následně odpovídá stejnou zprávou.

Konfigurační blok

V konfiguračním bloku může klientská aplikace číst a nastavovat parametry serveru. Výměna zpráv uvnitř konfiguračního bloku je založena na modelu *Request/Response*. Pro požadavky ze strany klientské aplikace slouží pouze následující dvě zprávy:

```
<pbset atribut="hodnota"/>
<pbget atribut=""/>
```

Pomocí elementu `pbset` může klient žádat server o nastavení atributu na zadanou hodnotu. Odesláním elementu `pbget` klient žádá server o zaslání nastavení zadaného atributu. Počet atributů uvnitř elementů není nijak omezen. U zprávy `pbget` jsou uvedeny pouze názvy atributů a hodnoty zůstávají nevyplněny. Na oba typy požadavků server odpovídá zprávou:

```
<pbans atribut="hodnota"/>
```

Zpráva odpovědi `pbans` musí vždy obsahovat stejný počet a typ atributů jako předchozí zpráva požadavku. Pokud server vrátí jiný počet atributů, nebo některý atribut neodpovídá požadavku, pak klient ukončí komunikaci. To samé platí i pro případ, kdy je přijat neznámý atribut nebo hodnota atributu není platná.

Následující blok obsahuje posloupnost zpráv, pomocí nichž dojde ke sjednocení nastavení všech parametrů na straně klienta a serveru.

```
<pbget baudrate=""/>
<pbans baudrate="187500"/>
<pbset filteron="0" triggeron="0"/>
<pbans filteron="0" triggeron="0"/>
```

Parametr `baudrate` specifikuje přenosovou rychlost sběrnice Profibus. Další dva parametry slouží k aktivaci/deaktivaci filtrace dat a podmíněného záznamu dat (*Triggeringu*). Další parametry přibudou po dokončení implementace zmiňovaných funkcí na straně serveru. Po provedení všech potřebných nastavení lze ukončit konfigurační blok odesláním ukončovacího elementu `</pbconfig>`. Server odpovídá stejným způsobem. Do konfigurační sekce se lze opět vrátit a opakovat nastavení parametrů.

Blok přenosu dat

Přenos dat ze serveru je aktivován pouze pokud uživatel stiskne tlačítko s nápisem **Data** v hlavním okně aplikace. Tlačítko je umístěno v dolní čísti panelu s informacemi o záznamu (viz obrázek 5.2). Po stisku tlačítka je nejprve vyprázdněna tabulka s daty a následně klient odešle serveru zprávu:

```
<pbdata count="100000"/>
```

Parametr `count` není povinný a udává, kolik rámců má být ze sběrnice Profibus nasnímáno a odesláno klientské aplikaci. Pokud není parametr uveden, tak jsou data odesílána, dokud klient nepošle zprávu `<pbdatastop/>`. Přenos dat může uživatel kdykoli ukončit stiskem stejného tlačítka, kterým byl přenos spuštěn. Po odeslání požadovaného množství rámců nebo po příchodu zprávy `<pbdatastop/>` server uzavře datový blok odesláním ukončovacího elementu `</pbdata>`. Data ze serveru jsou odesílána po blocích jako následující elementy:

```
<d>Data ze serveru</d>
```

Formát dat vnořených v uvedeném elementu popisuji v podkapitole 6.5. Přenos dat může být během jedné relace mezi klientem a serverem opakován. Celý komunikační proces dobře vystihuje schéma stavového automatu, které uvádím na obrázku 6.4.

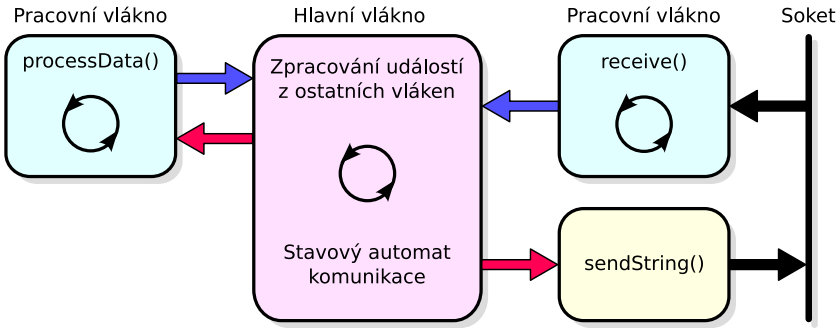
Ukončení spojení

Klientská aplikace může kdykoli ukončit spojení zasláním uzavíracího tagu kořenového elementu. Důvodem ukončení spojení může být požadavek uživatele, špatná syntaxe, obsah nebo pořadí zpráv. Klientskou aplikaci jsem naprogramoval tak, aby se vždy pokusila odeslat všechny potřebné ukončovací značky XML dokumentu. Pro tyto účely používám metodu `sendDisconnect()` umístěnou ve třídě `ComModule`.

Protokol prozatím nemá definovanou zprávu pro předávání důvodu ukončení spojení. V příští verzi aplikace plánuji přidání elementu s názvem `reason`, který by měl tento nedostatek odstranit.

6.4 Implementace obsluhy komunikace

V úvodu kapitoly jsem popsal postup pro navázání TCP spojení pomocí soketu. Nyní doplním informace o způsobu, jakým zpracovávám přijaté zprávy. V aplikaci používám vícevláknový model typu *Boss-Worker* (viz obrázek 6.3).



Obrázek 6.3: Schéma vícevláknového modelu pro obsluhu komunikace

Pracovní vlákno s metodou `receive()` zpracovává zprávy přijaté ze soketu a generuje události do hlavního vlákna. V kódu hlavního vlákna je implementován stavový automat, který podle aktuálního stavu a typu události rozhoduje o přechodu mezi stavy. Při chybě je komunikace korektně ukončena odesláním příslušných zpráv pomocí metody `sendString()`. Pokud jsou přijata binární data, tak je hlavní vlákno přes sdílenou paměť předá pracovnímu vláknu s metodu `processData()`. Po zpracování dat vygeneruje pracovní vlákno událost zpět do hlavního vlákna. Červené šipky tedy označují volání metod a modré šipky komunikaci pomocí událostí.

Zpracování XML z proudu dat je poněkud odlišné od zpracování běžného XML dokumentu. Dokument přichází skrz komunikační kanál jako posloupnost bajtů, ve které je nutné hledat otevírací a uzavírací značky jednotlivých elementů. Bajty přicházejí postupně v různém počtu, a proto musí být i XML dokument zpracováván postupně. Pro tyto účely lze použít tzv. SAX (*Simple API for XML*) parsery, které postupují sekvenčně podle množství nově příchozích bajtů. Tomuto způsobu zpracování XML se také říká progresivní parsování. Existují dva typy parserů.

Push parser čte postupně data z proudu a detekuje v nich značky elementů. Pokud je nalezena nějaká značka, tak parser generuje příslušnou událost. Druhou alternativou je tzv. *Pull* parser. Ten také detekuje jednotlivé značky v proudu dat, ale negeneruje žádné události. Místo toho čeká, až si uživatel detekovanou značku přečte pomocí některé z implementovaných metod. Pokud ještě není další element k dispozici, tak je volání metody pozastaveno, dokud nepřijde dostatek dat. Další informace o SAX parserech lze nalézt na internetových stránkách konzorcia W3C [11].

Platforma .NET Framework obsahuje ve jmenném prostoru `System.XML` třídu s názvem `XmlTextReader`. Instance této třídy pracuje jako pull parser, ale lze k němu jednoduchým způsobem doplnit události a vytvořit tak push parser. Zmíněný přístup používám v metodě `receive()`, jejíž obsah nyní stručně popíšu.

```
private void receive()
{
   try
   {
      XmlTextReader xmlReader = new XmlTextReader(nsRead);
      xmlReader.WhitespaceHandling = WhitespaceHandling.None;

      while (xmlReader.Read())
      {
         switch(xmlReader.NodeType)
         {
            case XmlNodeType.XmlDeclaration:
               ... Zpracování deklarační hlavičky XML dokumentu.
               break;

            case XmlNodeType.Element:
               ... Zpracování elementů dokumentu
               break;
         }
      }
   }
```

```
   // Ošetření výjimek.
   catch (IOException ioe) ...
   catch (XmlException xmle) ...
   catch (ThreadAbortException tae) ...
}
```

Z hlediska přehlednosti a úspory místa uvádím jen část zdrojového kódu. Na začátku metody inicializuji instanci třídy `XmlTextReader`, které je předána reference na proud dat ze soketu (`nsRead`). Dále následuje hlavní cyklus metody. Tělo cyklu se vykonává, dokud jsou k dispozici data ze soketu nebo dokud není generována nějaká výjimka. Voláním metody `Read()` se `xmlReader` posouvá na další přijatý element a data předchozího jsou zapomenuta. Pokud není další element k dispozici, tak se zpracování kódu programu zablokuje, dokud nepřijde dostatečné množství dat. Proto je také vhodné, aby metoda `receive()` nebyla vykonávána v hlavním vlákně aplikace.

Blok `switch` (přepínač) vykonává příslušný úsek kódu podle typu elementu načteného v `xmlReaderu`. Pro čtení dat z instance třídy `XmlTextReader` je definováno velké množství metod a vlastností. Zde uvádím ty, které v kódu používám:

Název vlastnosti/metody	Popis funkce
`AttributeCount`	Vlastnost vrací počet atributů uvnitř elementů.
`MoveToFirstAttribute()`	Posune `xmlReader` na první atribut elementu.
`MoveToNextAttribute()`	Posune `xmlReader` na další atribut elementu.
`Depth`	Udává hloubku vnoření elementu uvnitř dokumentu.
`ReadString()`	Přečte text vnořený v elementu.
`Name`	Vlastnost vrací název elementu nebo atributu.
`Value`	Vlastnost vrací obsah elementu nebo atributu.

Tabulka 6.2: Popis vybraných vlastností a metod třídy `XmlTextReader`

Pro předávání obsahu zpracovaných zpráv z pracovního do hlavního vlákna používám událost `ComEvent(element)`. Jako argument události je předávána instance třídy `Element`. Tato třída může obsahovat název, hodnotu a případně i dvourozměrné pole s atributy přijaté zprávy. Při vzniku chyby je v elementu předán název chyby. Pro předávání binárních dat jsem do třídy `Element` vložil ještě pole bajtů.

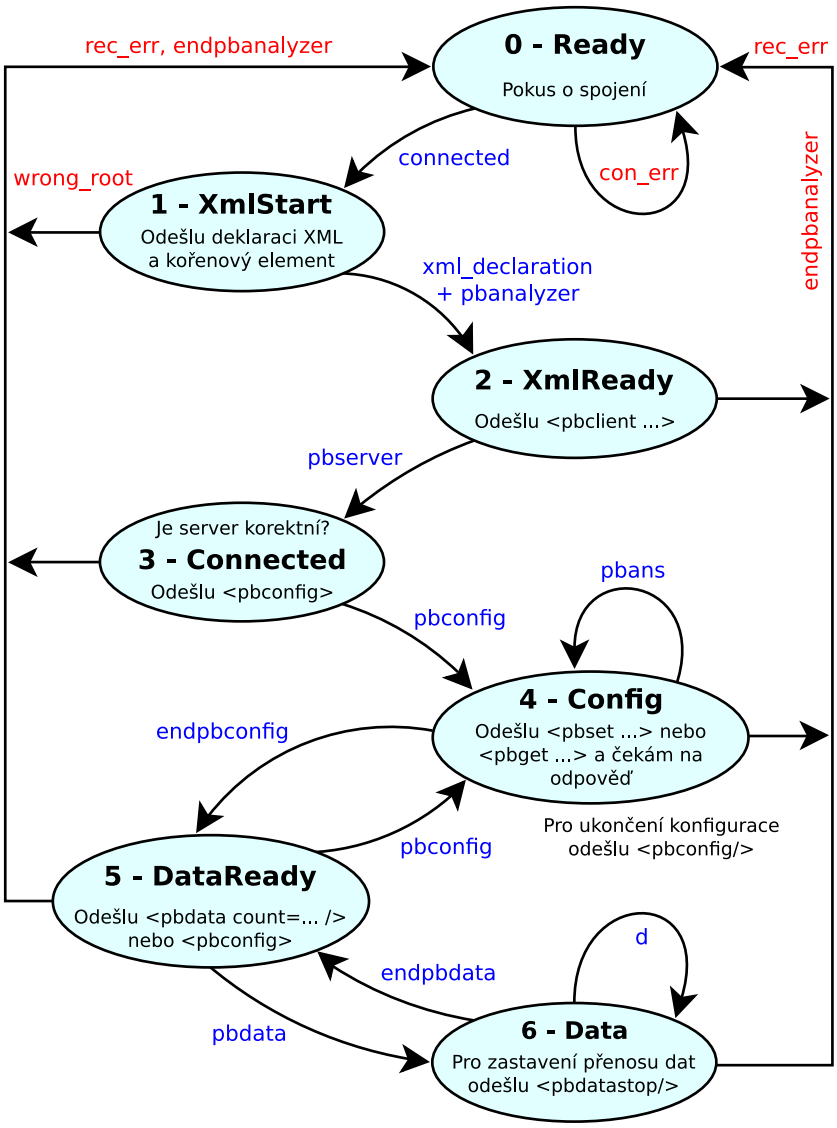
Událost `ComEvent` v hlavním vlákně obsluhuje metoda `parseComEvents(element)`. Metoda je umístěna ve třídě `ComModule` a obsahuje implementaci stavového automatu komunikace. Zde uvádím zkrácený výpis zmíněné metody:

```
private void parseComEvents(Element element)
{
   if (mainForm.InvokeRequired)
   {
      ComEventD ced = new ComEventD(parseComEvents);
      mainForm.Invoke(ced, new object[] { element });
   }
   else  // Zpracování obsahu události pro stavy automatu.
   {
      switch (element.name)
      {
         case ComStates.CONNECTED:
            msgConnected(element);
            break;

         case ComStates.DECLARATION_XML:
            RecElementEvent(element);
            break;

         case ComStates.XML_ROOT_NODE:
            msgRootNode(element);
            break;
         ...
      }
   }
}
```

Událost `ComEvent` je vyvolána v pracovním vlákně, a tudíž by byl i kód metody `parseComEvents` vykonáván v kontextu tohoto vlákna. Proto na začátku uvedeného kódu provádím pomocí metody `Invoke()` přepnutí kontextu do hlavního vlákna.

Následuje blok `switch`, ve kterém jsou podle jména zprávy (`element.name`) volány příslušné metody pro obsluhu události. Názvy jednotlivých zpráv jsou předefinovány ve třídě `ComStates`. V instanci této třídy je také uchováván aktuální stav stavového automatu. Podle aktuálního stavu komunikace a argumentu události se provádí kontrola sémantiky komunikačního protokolu. Při chybě je zavolána metoda `sendDisconnect()` a spojení se ukončí.

Kompletní schéma stavů a přechodů uvádím na obrázku 6.4. Uzly grafu obsahují názvy stavů a zprávy, které může odeslat klientská aplikace. Hrany grafu specifikují zprávy předávané při události `ComEvents` z metody `receive()`.



Obrázek 6.4: Stavový diagram komunikace na straně klientské aplikace

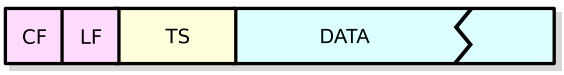
Záměrně jsem názvy zpráv volil v souladu s názvy elementů, aby byl graf srozumitelný. Ukončovací elementy mají místo / předponu `end`. Stavový automat se automaticky nastavuje na výchozí stav **Ready** při chybě v komunikačním protokolu nebo při ukončení spojení.

6.5 Přenos binárních dat

Původně jsem předpokládal, že bude stačit binární data ze serveru vnořit do XML elementu jako sekci CDATA. Sekce je uvozena počáteční značkou `<![CDATA[` a koncovou značkou `]]>`. Data, která jsou umístěna mezi uvedenými značkami, XML parser nezpracovává a předává je v nezměněné podobě. Později jsem však zjistil, že při výskytu netisknutelného znaku v sekci CDATA parser generuje výjimku a uzavře celý soket. Bylo tedy nutné nalézt jiný způsob, jak přenášet binární data a přitom zachovat XML protokol. Dříve, než začnu popisovat způsob přenášených dat, zmíním ještě par slov o jejich formátu.

6.5.1 Formát dat na serveru

Přípustné formáty rámců na úrovni linkové vrstvy Profibusu jsem popsal v sekci 2.2.2. Pro analýzu dat je však třeba znát i čas, kdy byl každý rámec přijat. Pro tyto účely je na straně serveru doplněna ke každému rámci hlavička s časovou značkou. Výslednou strukturu rámce ukazuje následující obrázek 6.5.



Obrázek 6.5: Formát dat na serveru

Pole **CF** (*Control Field*) má délku 1 bajt a obsahuje informaci o délce pole s časovou značkou **TS** (*Time Stamp*). Přiřazení jednotlivých délek TS k hodnotám bajtu CF uvádí tabulka 6.3. Pole s označením **LF** (*Length Field*) obsahuje právě jeden bajt, jehož hodnota odpovídá délce sekce DATA.

Hodnota CF	Délka TS
0	1 B
1	2 B
2	4 B
3	8 B

Tabulka 6.3: Délka časové značky podle CF.

Z hodnot pole CF vyplývá, že jsou použity pouze dva bity s nejnižším významem. V budoucnu by tedy mohl být význam CF rozšířen o další možnosti. Pole TS obsahuje časovou značku vyjádřenou v jednotkách t_{bit}, které jsem uvedl již v kapitole o Profibusu 2. Převod hodnoty TS na čas lze provést vynásobením převrácenou hodnotou přenosové rychlosti. Jednotlivé bajty TS jsou řazeny v pořadí *Little Endien*. Sekce DATA obsahuje vždy jeden rámec ze sběrnice Profibus. Délka pole DATA je tedy v rozsahu 1 – 255 bajtů.

6.5.2 Řešení přenosu binárních dat

Pro řešení problému s přenosem binárních dat jsem navrhl následující 4 možnosti:

- Hexadecimální reprezentace dat
- Použití kódování Base64
- Odpojování XML parseru
- Komunikace pomocí dvou soketů

Hexadecimální reprezentace dat

Binární data je možné transformovat na text. Každý bajt je v textovém řetězci reprezentován dvojicí znaků. Jednotlivé znaky mohou obsahovat pouze číslice 0 – 9 a písmena A – F. Takto reprezentovaná data lze předávat uvnitř běžného XML elementu jako text.

Nevýhodou tohoto způsobu je zdvojnásobení objemu přenášených dat. Implementace zpracování těchto dat je velice snadná a výpočetně nenáročná. Třída `XmlTextReader` přímo obsahuje metody pro čtení takto reprezentovaných dat.

Použití kódování Base64

Kódování Base64 slouží pro převod obecných binárních dat na posloupnost tisknutelných znaků. Používá se například pro přenos obrázků uvnitř emailové zprávy. Visual Studio tímto způsobem ukládá obrázky do XML souborů prostředků.

Princip kódování je relativně jednoduchý. Na obrázku 6.6 zobrazuji zakódování rámce tokenu. Algoritmus postupně převádí každou trojici bajtů vstupních dat na čtveřici tisknutelných znaků z následující kódové abecedy.

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
```

Data	0xDC								0x02								0x06							
Bitový zápis	1	1	0	1	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1	0
Index	55						0						8				6							
Base64 kód	3						A						I				G							

Obrázek 6.6: Ilustrace kódování Base64

Každou trojici vstupních bajtů lze vyjádřit jako posloupnost 24 bitů. Posloupnost je následně rozdělena do bloků po šesti bitech a vznikne tak čtveřice indexů. Dekadická hodnota indexu pak přímo ukazuje na jeden ze znaků v kódové abecedě. Na výstupu algoritmu tedy dostaneme čtyři tisknutelné znaky z kódové abecedy.

Pokud na konci vstupních dat nejsou k dispozici 3 bajty, tak je posloupnost bitů automaticky doplněna nulami. V zakódovaných datech se následně objeví jeden nebo dva znaky „=“, protože délka výstupního textového řetězce musí být dělitelná čtyřmi bezezbytku. Při zpětné transformaci dekodér podle počtu znaků „=“ pozná kolik bajtů bylo doplněno nulami a odstraní je.

Z popisu algoritmu vyplývá, že se nárůst objemu přenášených dat pohybuje okolo 33 %. Implementace dekódování dat není nijak náročná, protože `XmlTextReader` přímo obsahuje metodu `ReadContentAsBase64()`, které se pouze předá pole bajtů inicializované na potřebnou velkost. Vhodnější je metoda `System.Convert.FromBase64String()`, která rovnou vrací pole dekódovaných dat bez nutnosti inicializace pole. Další podrobnosti o kódování Base64 jsou k dispozici na webových stránkách [15].

Odpojování XML parseru

Toto řešení se týká hlavně klientské aplikace. Po vytvoření komunikačního kanálu probíhá běžná výměna konfigurace podle výše popsaného protokolu. Na začátku přenosu binárních dat se v klientské aplikaci odpojí XML parser od soketu. Tím je umožněno ze serveru posílat data bez dalšího kódování. Po příjmu požadovaného množství rámců klient ukončí přenos binárních dat zasláním příslušné zprávy serveru. Následně by se opět připojil XML parser a mohla by pokračovat komunikace podle XML protokolu.

Výhodou popisovaného řešení je nulový nárůst objemu přenášených dat. Po dobu odpojení parseru však klient nemůže přijímat žádné řídicí zprávy ze serveru. Implementace tohoto řešení skončila na problému s vyprazdňováním vyrovnávací paměti soketu před opětovným připojením XML Parseru. Pokud vyrovnávací paměť obsahovala nějaká data, tak došlo k ukončení spojení.

Komunikace pomocí dvou soketů

Poslední možností je otevření dvou komunikačních soketů. První soket pak slouží pro předávání konfigurace a informačních zpráv podle protokolu definovaného v sekci 6.3. Pro přenos binárních dat je vyhrazen druhý soket.

Stejně jako u předchozího řešení nedochází k navýšení objemu dat, ale navíc je v každém okamžiku umožněna obousměrná komunikace mezi klientem a serverem. Velkou nevýhodou tohoto řešení je nutnost přepracování části zdrojových kódů na straně klientské i serverové aplikace. Je nutné doplnit další vlákna a prostředky pro správu druhého soketu.

Po konzultaci zmíněných řešení s vedoucím diplomové práce a Martinem Samkem, který vyvíjí serverovou aplikaci, jsem zvolil variantu využívající Base64 kódování. Pro první verzi aplikace je toto řešení postačující. V budoucí verzi programu se počítá s použitím dvou komunikačních soketů.

Kapitola 7

Zpracování a zobrazení dat

Účelem této práce je návrh a ověření funkce základních prostředků pro zpracování a zobrazení dat. V následujících verzích programu budou zmiňované prostředky dále vylepšovány, aby bylo dosaženo maximálního výkonu. V první části kapitoly se věnuji zpracování dat a způsobu jejich uchovávání v paměti počítače. V druhé části se následně zabývám zobrazením dat a doplňkovými funkcemi pro jejich interpretaci. Příslušné zdrojové kódy jsem uložil do jmenného prostoru `DataProcessing`, jehož obsah uvádím v následujícím seznamu.

- `FrameInterpretation` – složka obsahuje zdrojové kódy pro interpretaci obsahu rámců sběrnice Profibus podle protokolu DP.
- `Frames` – zde jsou uloženy třídy pro uchovávání dat v paměti počítače.
- `Records` – složka obsahuje třídy pro uchovávání informací o záznamu.
- `Resources` – složka se soubory prostředků pro jazykovou lokalizaci.
- `Statistics` – zde jsou uloženy zdrojové kódy tříd pro výpočet statistik.

7.1 Uchovávání dat v paměti počítače

Do aplikace lze načítat data ze souboru záznamu uloženého na disku nebo přímo ze vzdáleného serveru. V obou případech je potřeba uchovávat zpracovaná data v paměti počítače, aby mohla být zobrazena a dále interpretována. V první verzi programu všechny zpracované rámce záznamu uchovávám naráz v operační paměti. V praxi se však ukázalo,

že pro načtení 10-ti milionů rámců je potřeba cca 950 MB operační paměti. Proto jsem do hlavního nastavení aplikace přidal prvky pro konfiguraci maximálního počtu zpracovávaných rámců. Nejvyšší nastavitelná hodnota je právě 10 milionů. Potřebné hardwarové prostředky pro zpracování tohoto počtu rámců uvádím v příloze A.

V příštích verzích programu je počítáno s doplněním stránkování dat, čímž se odstraní velké paměťové nároky a zároveň i omezení celkového počtu zpracovávaných rámců. Jednu z možných verzí stránkování popisuji v závěru této práce.

Data z binárního souboru i ze vzdáleného serveru obsahují rámce, jejichž formát jsem specifikoval v sekci 6.5.1. Data reprezentovaná jako řetězce bajtů však nejsou vhodná pro další zpracování. Pro uchovávání dat v operační paměti proto používám instanci třídy `PB_Frame_List`, ve které jsou uloženy jednotlivé rámce jako objekty typu `PB_Frame`. Obě zmíněné třídy jsou dostupné ve jmenném prostoru `DataProcessing.Frames`.

Konstruktoru třídy `PB_Frame` předávám pořadové číslo rámce, časovou značku a pole bajtů obsahující právě jeden rámec FDL vrstvy (viz formáty rámců uvedené v sekci 2.2.2). Všechny prvky předané konstruktoru jsou uvnitř třídy uloženy pro další použití. Konstruktor následně rozebírá a validuje FDL rámec. V každém objektu typy `PB_Frame` je tedy navíc uložena zdrojová a cílová adresa s příslušnými přístupovými body (SAPy).

Dále je z bajtu s označením FC (viz sekce 2.2.2) zjištěn typ FDL služby. Každé FDL službě jsem v aplikaci přiřadil unikátní celé číslo. Kompletní výčet FDL služeb je k dispozici ve specifikaci Profibusu [2, Part 4, kapitola 4.7.3]. Podle typu služby a přístupových bodů (SAPů) se následně určuje, zda se jedná o zprávu protokolu Profibus DP. Přiřazení typu DP zprávy k příslušné FDL službě a SAPům zobrazuje tabulka 2.1 v sekci s popisem uživatelské vrstvy Profibusu 2.3.1. Podobně jako u služeb jsem každému typu DP zprávy přiřadil unikátní celočíselný identifikátor. Pro čtení obsahu jednotlivých částí rámce jsem ve třídě `PB_Frame` implementoval potřebné vlastnosti (*Properties*).

Třída `PB_Frame_List` je zděděna od třídy `BindingList` a slouží jako kolekce objektů typu `PB_Frame`. Uvedená rodičovská třída má implementováno rozhraní `IList`, které umožňuje indexovat jednotlivé objekty uvnitř kolekce. Dále poskytuje metody pro přidávání a odstranění položek kolekce (`Add()`, `Remove()`, `Clear()`). Díky dědičnosti má vyjmenované možnosti i třída `PB_Frame_List`. `BindingList` používám jako rodičovskou třídu také proto, že je podporována jako zdroj dat u komponenty `DataGridView`. V sekci 7.4.1 se zmiňuji o využití uvedené vlastnosti pro zobrazování dat.

7.2 Zpracování dat ze záznamu

Základní popis záznamu a způsob jeho ukládání do dvou souborů jsem uvedl v podkapitole 6.1. Nyní doplním informace o samotném načítání binárních dat záznamu do operační paměti. Pro otevření záznamu slouží v hlavním menu aplikace položka **Soubor → Otevřít**. Po jejím zvolení musí uživatel v dialogovém okně zadat název a cestu k souboru s informacemi o záznamu (`.rcd`). Po potvrzení volby se provede deserializace obsahu souboru do objektu typu `Record`. Postup deserializace zde již neuvádím, protože je stejný jako u souboru s nastavením (viz sekce 4.1.2).

Při úspěšném provedení deserializace se aplikace pokouší otevřít soubor s binárními daty, jehož název je uveden v načtených informacích o záznamu. Vždy se předpokládá, že je binární soubor umístěn ve stejném adresáři jako soubor s příponou `.rcd`. Pokud není binární soubor nalezen, tak je signalizována chyba a celý záznam se automaticky uzavře. V opačném případě je vytvořeno a spuštěno nové vlákno, které vykonává metodu `readBinFile()` s následujícím obsahem.

```
 public void readBinFile()
 {
    FileStream fs = new FileStream(...);
    DataReader dataReader = new DataReader(globalSettings, fs);

    pb_Data.Set_PB_Frames(dataReader.readAndParseDataFromStream());
    fs.Close();
    dataReader = null;

    onReadBinFileEnd;
}
```

Nejprve je otevřen soubor s binárními daty a vytvořena instance třídy `DataReader`, jež obsahuje metody pro zpracování dat. Metoda `readAndParseDataFromStream()` postupně čte data ze souboru a zpracovává jednotlivé rámce ve formátu uvedeném na obrázku 6.5. Nejdříve je načtena hlavička s časovou známkou rámce a následně je vytvořena nová instance třídy `PB_Frame`, která je přidána do kolekce typu `PB_Frame_List`. Celý postup se opakuje, dokud není zpracován celý obsah souboru nebo dokud není načteno množství rámců zadané na formuláři `SettingsForm`.

Po ukončení zpracování dat metoda `readBinFile()` uzavře binární soubor a generuje událost `onReadBinFileEnd`. V hlavním vlákně aplikace je po vzniku události zavolána metoda `readBinFileEnd()` starající se o zobrazení dat.

7.3 Zpracování dat ze serveru

V kapitole o komunikaci 6 jsem na obrázku 6.3 uvedl vícevláknový model aplikace. Popsal jsem způsob zpracování přijatých zpráv a předávání dat mezi jednotlivými vlákny.

O příjem zpráv z TCP soketu se stará pracovní vlákno vykonávající kód metody `receive()`. Zpracování binárních dat přijatých ze serveru provádí druhé pracovní vlákno v metodě `DataReader.processData()`. Uvedené vlákno je vytvořeno při aktivaci režimu čtení dat ze serveru. V metodě `processData()` je nejprve vytvořena instance třídy `ArrayList` sloužící jako mezipaměť (*buffer*) pro přijatá data. Dále jsou odstraněna předchozí zpracovaná data z kolekce typu `PB_Frame_List` umístěné v aktuálním záznamu[1]. Pro ukládání přijatých dat je v pracovním adresáři[2] vytvořen nový soubor s názvem `temp.bin`. Pokud již soubor s tímto názvem existuje, tak je přepsán novým. Po provedení inicializace je pracovní vlákno uspáno a čeká na data.

Přijaté zprávy s binárními daty zakódovanými pomocí algoritmu Base64 jsou v metodě `receive()` dekódovány do pole bajtů. V tomto poli jsou uloženy rámce ve formátu uvedeném na obrázku 6.5. Následně je vyvolána událost `ComEvent(element)`. V argumentu události je pole bajtů předáno hlavnímu vláknu a dále do výše popsaného bufferu vlákna pro zpracování dat. Pracovní vlákno je probuzeno zavoláním metody `Interrupt()`.

Každé pole bajtů vyzvednuté z bufferu je postupně procházeno a rozebíráno metodou `readAndParseDataFromArray()`. U jednotlivých rámců je nejprve zpracována hlavička s časovou značkou a následně se vytváří nová instanci třídy `PB_Frame`. Konstruktoru třídy se předávají všechny potřebné údaje rámce. Nakonec je instance třídy `PB_Frame` vložena do kolekce typu `PB_Frame_List`. Po zpracování všech rámců je pole bajtů přidáno do binárního souboru `temp.bin`.

Zpracování dat probíhá, dokud jsou k dispozici nová data v bufferu vlákna. Pokud je buffer prázdný, tak se vlákno uspí a čeká na další data. Datový buffer slouží jako tzv. *sdílená paměť*, a proto musí být pro zápis a čtení dat použity synchronizační prostředky. V klientské aplikaci pro tyto účely používám jednoduchý zámek (**mutex**). Podrobnosti o synchronizačních prostředcích platformy .NET lze nalézt v knize [6, Kapitola 5].

[1] Pojmem záznam je myšlena instance třídy `Record`

[2] Pracovní adresář lze zvolit na formuláři nastavení `SettingsForm`

7.4 Zobrazení dat

Pro zobrazování dat v tabulkové podobě .NET Framework poskytuje několik komponent. Nejvhodnější z nich jsou `ListView` a `DataGridView`. Pro klientskou aplikaci jsem zvolil druhou jmenovanou komponentu, protože je na rozdíl od `ListView` daleko flexibilnější a poskytuje mnoho možností nastavení. Mezi užitečné funkce patří například barevné zvýrazňování řádek, sloupců a buněk pomocí stylů.

7.4.1 Komponenta DataGridView

V této práci není prostor na podrobný popis všech možností komponenty `DataGridView`. Podrobněji proto uvádím pouze dva způsoby zobrazování dat použité při vývoji klientské aplikace. Další informace jsou k dispozici v literatuře [13, Chapter 15].

Komponentu typu `DataGridView` s názvem `dataTable` jsem umístil do datového panelu hlavního okna aplikace. Na obrázku 5.2 je panel označen číslem 2. Před samotným zobrazením dat je nutné ručně nadefinovat jednotlivé sloupce viditelné v komponentě `dataTable`.

Přidání a nastavení sloupečků tabulky lze provést přímo v návrháři vývojového prostředí VS. V okně **Properties** komponenty `dataTable` stačí otevřít editor u vlastnosti `Columns`. Do tabulky jsem pomocí uvedeného editoru přidal příslušný počet sloupečků pro zobrazení informací obsažených v instanci třídy `PB_Frame`. U každého sloupečku jsem nastavil název viditelný v záhlaví tabulky, šířku sloupce, jeho pořadí a zarovnání textu.

Uživatel může skrývat sloupečky a měnit jejich pořadí. Nastavení je dostupné na jedné ze stránek formuláře `SettingsForm`. Viditelné sloupečky lze také přímo vybrat v kontextovém menu záhlaví tabulky. Konfigurace sloupečků je při ukončování programu ukládána do instance třídy `DataSettings` uvnitř hlavního nastavení aplikace.

Komponenta typu `DataGridView` umožňuje načítat data několika způsoby. V počáteční verzi programu jsem používal tzv. *Custom Data Binding*. Později jsem však z důvodu úspory paměti a urychlení zobrazení dat přešel na tzv. *Virtuální mód*.

Custom Data Binding

Použití tohoto způsobu načítání dat do komponenty `dataTable` je relativně jednoduché. Obsah jednotlivých sloupců tabulky je naplněn pomocí vlastností, které jsou implementovány ve třídě `PB_Frame`. Ve výše uvedeném editoru pro konfiguraci sloupečků stačí přiřadit jako hodnotu `DataPropertyName` název příslušné vlastnosti ze třídy `PB_Frame`. Tímto

způsobem lze velmi efektivně určit, jak budou data obsažená v instancích třídy `PB_Frame` zobrazena v tabulce. Vlastní načtení dat do tabulky se provede jednoduchým přiřazením reference na instanci třídy `PB_Frame_List` do vlastnosti `DataSource` u komponenty `dataTable`. Následně se automaticky vytvoří reference mezi řádky tabulky a příslušnými instancemi třídy `PB_Frame`.

Popsaný postup je sice velmi efektivní, ale při velkém množství řádků tabulky je načítání dat značně pomalé a vyžaduje velké množství operační paměti.

Virtuální mód

Virtuální mód umožňuje rychleji zobrazit data a alokuje méně operační paměti. U komponenty typu `DataGridView` se virtuální mód aktivuje nastavením vlastnosti `VirtualMode` na hodnotu `true`. Pro zobrazení dat je dále nutné implementovat metodu pro obsluhu události `CellValueNeeded`. Uvedená událost je generována pokaždé, když je potřeba na obrazovce překreslit obsah nějaké buňky v tabulce. Z argumentů události lze zjistit řádkový a sloupcový index požadované buňky. Podle řádkového indexu je možné přímo identifikovat instanci třídy `PB_Frame` uvnitř kolekce typu `PB_Frame_List`. Podle sloupcového indexu je následně zavolána příslušná vlastnost pro naplnění buňky v žádaném sloupci.

Pro zobrazení dat v tabulce nyní stačí do vlastnosti `dataTable.RowCount` zadat celkový počet řádků tabulky a metoda obsluhující událost `CellValueNeeded` se postará o vše potřebné. Oproti předchozímu způsobu zobrazení dat vyžaduje virtuální mód minimální množství paměti, protože jsou vždy přiřazována data pouze viditelným řádkům tabulky. Výsledek po načtení dat do tabulky zobrazuje obrázek 7.1.

Datový panel

	Číslo rámce	Časová značka	Formát rámce	Adresy	SAPy	Funkce (FC)
	243	0,044 ms	SD1	0 --> 6		49
	244	0,066 ms	SD4	0 --> 0		
	245	0,044 ms	SD1	0 --> 7		49
	246	0,066 ms	SD2	0 --> 3	62 --> 62	7d
	247	0,249 ms	ACK			
	248	0,029 ms	SD2	0 --> 4	62 --> 62	7d
	249	0,110 ms	ACK			
	250	0,029 ms	SD2	0 --> 16	62 --> 62	7d
	251	0,117 ms	ACK			
	252	0,029 ms	SD4	0 --> 0		

Obrázek 7.1: Datový panel hlavního okna aplikace

7.4.2 Interpretace dat podle specifikace Profibus DP

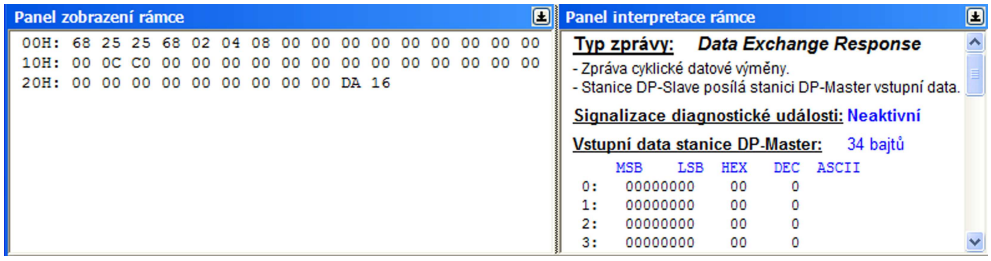
V kapitole 5 jsem popsal jednotlivé panely uvnitř hlavního okna aplikace. Zmínil jsem se také o panelech pro zobrazení obsahu a interpretaci rámce, v nichž je vložena komponenta typu `RichTextBox`. Nyní podrobněji popisuji obsah těchto panelů.

Při každém označení nebo změně výběru buňky v datové tabulce je vygenerována událost `SelectionChanged`, jejíž argumenty obsahují řádkový index vybrané buňky. Následně je v datové kolekci typu `PB_Frame_List` nalezena instance třídy `PB_Frame` odpovídající řádkovému indexu. Do panelu určeného pro zobrazení obsahu rámce je vložen textový výpis vybraného rámce. Na obrázku 7.2 je zobrazen výsledný obsah panelu.

Pro naplnění panelu s interpretací rámce je nejprve vytvořena nová instance třídy `FrameInterpretation`. Následně je zavolána metoda `setFrameInterpretation(...)`, které je jako argument předána příslušná instance třídy `PB_Frame`. V první fázi je zjištěn typ zprávy. Pokud se jedná o zprávu protokolu Profibus DP, tak je podle specifikace uživatelské vrstvy[3] rozebrán datový obsah celého rámce. Do panelu pro interpretaci rámce je následně vložen textový dokument ve formátu RTF popisující význam datové části rámce.

RTF (*Rich Text Format*) je velmi jednoduchý značkovací jazyk pro formátování textu. V hlavičce RTF dokumentu jsou definovány jednotlivé barvy a styly písem. V textu se na ně lze odkazovat pomocí značek. Výhodou tohoto přístupu je možnost snadné jazykové lokalizace popisu obsahu jednotlivých zpráv, protože části dokumentu lze i se značkami uložit do souboru s prostředky[4] jako obyčejný text.

Komponenta typu `RichTextBox` umí přímo zpracovávat RTF formát. Na obrázku 7.2 je zobrazen obsah panelu s výpisem interpretace zprávy typu *Data Exchange Response*. Podrobnosti o formátu RTF lze nalézt na internetových stránkách [16].



Obrázek 7.2: Obsah panelů s doplňkovými informacemi rámce

[3]Uživatelská vrstva je podrobně rozebrána ve specifikaci Profibusu [2, Part 8]

[4]Soubory prostředků jsem popsal v sekci 4.2.

7.5 Filtrace a podmíněný záznam dat

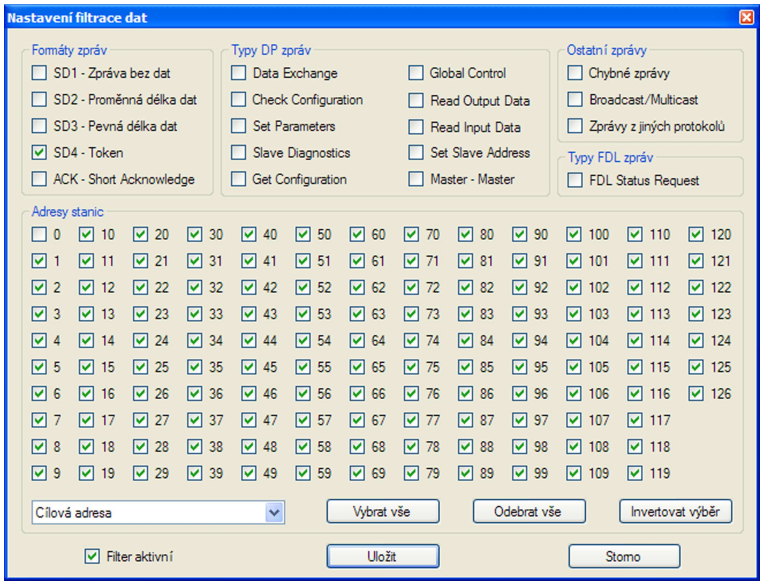
Datový záznam může obsahovat velké množství rámců ze sběrnice Profibus. Procházení všech rámců je pak velmi zdlouhavé a náročné. Běžné analyzátory komunikace jsou proto vybaveny funkcemi pro filtraci a podmíněné spouštění záznamu dat (*Trigger*). Obě zmíněné funkce přispívají k redukci množství zobrazovaných dat, a tím se zkracuje i čas potřebný pro jejich analýzu.

Analyzátor, který popisuji v této práci, je navržen tak, aby umožňoval dvouúrovňovou filtraci i podmíněné spouštění záznamu dat. Vývoj triggeru a první úrovně filtru není zatím na straně serverové aplikace dokončen. Druhou úroveň filtrace jsem implementoval ve své klientské aplikaci. Podrobnosti o jednotlivých funkcích a jejich realizaci uvádím v následující části této kapitoly.

7.5.1 Filtrace zobrazovaných dat

Většinu zdrojového kódu, pomocí kterého provádím filtraci zobrazovaných dat, jsem prozatím umístil přímo do třídy hlavního formuláře `MainForm`. Při budoucím rozšíření filtrace bude pravděpodobně vhodnější tento kód přesunout do samostatné třídy.

Pro nastavení filtru jsem vytvořil formulář `FilterFormu`, který zobrazuji na následujícím obrázku 7.3.



Obrázek 7.3: Formulář pro nastavení filtru zobrazovaných dat

Formulář lze zobrazit pomocí položky menu v hlavním okně aplikace (**Nástroje → Nastavení lokálního filtru**). Pro jednotlivé prvky nastavení jsem použil komponenty typu `CheckBox` uspořádané do skupin pomocí `GroupBoxů`.

Na formuláři umožňuji nastavení filtrace dat podle formátu rámce, adres a typu DP nebo FDL zprávy. Skupina s označením „Ostatní zprávy“ umožňuje dále filtrovat chybné rámce, rámce typu Broadcast/Multicast a zprávy dalších protokolů, které neodpovídají specifikaci Profibus DP.

V současné verzi aplikace lze uložit pouze jedno nastavení filtru. Při vývoji filtrace jsem se zabýval hlavně efektivním aplikováním filtru na data. V závěrečné kapitole 8 uvádím námět na možné rozšíření filtrace.

Nastavení filtru ukládám do instance třídy `FilterSettings` obsažené uvnitř výše popsané třídy `GlobalSettings` (viz tabulka 4.1). Konstruktoru formuláře `FilterForm` předávám instanci třídy `FilterSettings` a podle jejího obsahu se přednastavují prvky na formuláři. Nastavení filtrace zdrojové a cílové adresy lze provést ve skupině s označením „Adresy stanic“. Pro přepínání mezi zdrojovou a cílovou adresou slouží komponenta typu `ComboBox` (`cbSelectAddr`) v levé dolní části formuláře. Pomocí tří tlačítek lze následně označit nebo odznačit všechny adresy nebo invertovat výběr.

Při stisku tlačítka **Uložit** se uloží stav nastavení filtru z formuláře zpět do instance třídy `FilterSettings`. Při stisku tlačítka **Storno** se změny neukládají a formulář se pouze zavře.

7.5.1.1 Způsob aplikování filtru

Samotná komponenta typu `DataGridView` nepodporuje filtraci dat. Byl jsem tedy nucen implementovat vlastní metodu. Po zvážení různých řešení jsem nakonec zvolil metodu využívající vyhledávací tabulku (*Lookup Table*). Vytvořil jsem pole s položkami typu `int`, jehož délku inicializuji na celkový počet rámců načtených v instanci třídy `PB_Frame_List`. V aplikaci jsem pole pojmenoval `indices`. Po aktivaci filtru z formuláře pro nastavení filtrace nebo přímo z menu aplikace je zavolána metoda `applyFilter()`.

Metoda postupně prochází všechny rámce obsažené v kolekci `PB_Frames_List` a u každého z nich rozhoduje, zda má být zobrazen. Pokud rámec splňuje podmínky filtru, tak je jeho index přidán do vyhledávací tabulky `indices`. Proměnná `index` s informací o počtu položek v poli je následně navýšena o jedničku. Takto jsou zkontrolovány všechny rámce. Na konci metody `applyFilter()` je provedeno obnovení obsahu datové tabulky pomocí následujících příkazů:

```
dataTable.RowCount = 0;
dataTable.RowCount = index;
```

Nejprve je počet položek tabulky nastaven na 0, aby došlo k jejímu vymazání a následně je vložen počet položek zbylých po aplikování filtru. Tím dojde k obnovení obsahu komponeny `dataTable`. Při události `CellValueNeeded` jsou v tabulce zobrazovány rámce podle indexu z vyhledávací tabulky. Tím je zaručeno, že se budou v tabulce zobrazovat jen rámce splňující filtrační podmínku. Stejným způsobem je upravena i metoda ošetřující výše popsanou událost `SelectionChanged`.

Popsaný způsob filtrace sice vyžaduje alokaci určitého množství operační paměti, ale na druhou stranu je velmi rychlý. 10 milionů zpráv je přefiltrováno maximálně za 2 sekundy.

7.5.2 Filtrace a podmíněný záznam dat na serveru

Filtrace dat a podmíněné spouštění záznamu není v aktuální verzi serverové aplikace plně implementováno. Na straně klientské aplikace jsem proto provedl pouze přípravu na budoucí rozšíření.

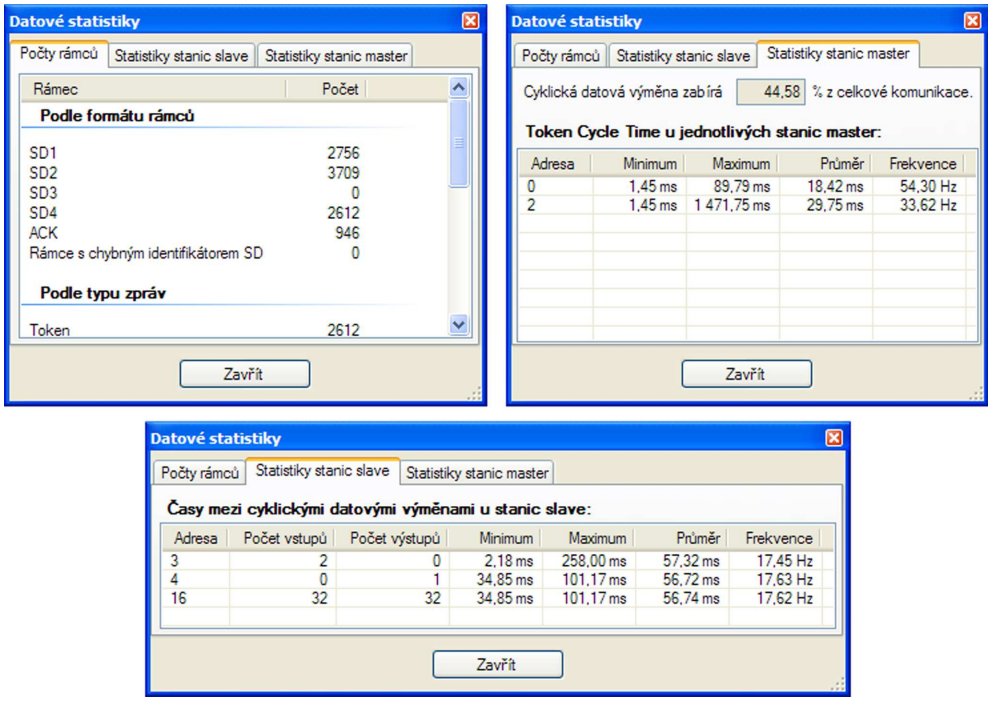
Pro aktivaci zmiňovaných funkcí bude potřeba rozšířit komunikační protokol o atributy pro předávání nastavení. Dále bude nutné na straně klientské aplikace doimplementovat položky pro ukládání nastavení a na formulář `ComConfigForm` přidat potřebné konfigurační komponenty.

Funkce podmíněného spouštění záznamu dat bude vhodná hlavně pro dlouhodobé monitorování komunikace na sběrnici Profibus. Princip funkce je velmi jednoduchý. V konfiguraci funkce je nastavena spouštěcí podmínka obsahující specifikaci typu zprávy a jejího obsahu. Pokud je nalezen rámec vyhovující zadané podmínce, je uloženo zadané množství dat před a za tímto rámcem. Tím se tedy významně redukuje počet ukládaných zpráv.

Filtrace dat na serveru přináší výhody i nevýhody. Mezi nesporné klady popisované funkce patří redukce objemu dat předávaných mezi serverovou a klientskou aplikací. Při odfiltrování části dat však nelze korektně vypočítat statistiky z nasnímaných dat na straně klientské aplikace.

7.6 Datové statistiky

Statistiky jsou vypočítávány z dat načtených ze souboru nebo ze vzdáleného serveru. Pro výpočet statistik jsem implementoval třídy `BasicStat` a `DevStat` umístěné ve jmenném prostoru `DataProcessing.Statistics`. Výsledky výpočtů jsou zobrazovány na formuláři `StatisticForm`.



Obrázek 7.4: Formulář pro zobrazení statistik

Statistiky lze zobrazit pomocí položky v hlavním menu aplikace (**Nástroje** → **Datové statistiky**). V první fázi je zavolán konstruktor formuláře `StatisticForm`, v němž se vytvoří instance třídy `BasicStat`. Následně se analyzuje obsah rámců typu `PB_Frame` uložených uvnitř kolekce `PB_Frame_List`. Výsledky statistik se ukládají v instanci třídy `BasicStat`. V závěru je zavolána metoda `showStatistisc()`, která se postará o zobrazení výsledků na formuláři.

Během procházení celé kolekce rámců je do instance třídy `BasicStat` ukládán počet výskytů jednotlivých formátů rámců a typů zpráv. Pro každou stanici DP Slave detekovanou z obsahu rámců je vytvořena nová instance třídy `DevStat`. V této třídě je uchovávána minimální a maximální doba mezi jednotlivými zprávami typu *Data Exchange Request*. Dále je uvnitř třídy uložen celkový čas a počet výskytů zmíněných zpráv, aby mohl být následně vypočítán průměrný čas mezi datovými výměnami dané stanice.

U stanic typu DP Master se stejným způsobem určuje doba oběhu tokenu. Pouze místo zprávy `Data Exchenge Request` je pro výpočty použita zpráva pro předávání *tokenu*. Na obrázku 7.4 je zobrazen obsah jednotlivých stránek formuláře `StatisticForm` po načtení výsledků výpočtů. Všechny časy jsou v instancích tříd `BasicStat` a `DevStat` uloženy v jednotkách t_{bit}. Při zobrazení na formuláři jsou následně přepočítávány na jednotky času.

Kapitola 8

Závěr

Cílem mojí diplomové práce byl návrh a implementace základních částí klientské aplikace pro analyzátor protokolu Profibus DP. Moje práce přímo navazuje na diplomovou práci kolegy Martina Samka, která se zabývá vývojem serverové aplikace.

Při implementaci programu jsem se zaměřil hlavně na vývoj základních částí pro příjem, zpracování a zobrazení dat přijatých ze vzdáleného serveru. Hlavní okno aplikace jsem rozčlenil na několik panelů, čímž jsem získal přehledné grafické rozhraní s jednoduchým ovládáním.

Program obsahuje komplexní systém pro ukládání konfigurace grafického rozhraní a zpracování dat. Uživatel tedy nemusí při každém spuštění programu znovu nastavovat všechny parametry. Grafické rozhraní aplikace je navrženo s ohledem na lokalizaci do dalších jazyků.

Současná verze programu umožňuje online čtení a ukládání dat ze vzdáleného serveru. Uložená data mohou být opětovně načtena a zobrazena v grafickém rozhraní. Jelikož současný způsob načítání dat má velké nároky na velikost operační paměti, musel jsem omezit maximální počet zpracovávaných rámců. Možné odstranění tohoto i dalších nedostatků popisuji v následující podkapitole.

Z doplňkových funkcí je implementováno základní rozhraní pro filtraci zobrazených dat a výpočet datových statistik. Komunikační protokol mezi klientskou a serverovou aplikací je otevřený pro budoucí rozšíření. Jedná se především o následující funkce: filtrace dat, podmíněné spouštění záznamu na straně serveru (*Trigger*) a offline čtení dat uložených na serveru.

Analyzátor funguje bez problémů pouze při nastavení přenosové rychlosti Profibusu na hodnotu menší než 500 kb/s. Při vyšší rychlosti začalo docházet k chybám přenosu, které pravděpodobně způsobuje XML Parser na straně klientské aplikace.

8.1 Vylepšení do budoucna

Budoucí vývoj analyzátoru by měl být zaměřen především na odstranění výše uvedených nedostatků a omezení. Zde uvádím několik námětů na možná vylepšení:

Stránkování dat - Nejslabším místem současné aplikace jsou velké nároky na velikost operační paměti a s tím související omezení maximálního počtu zpracovávaných rámců.

Problém by měl být odstraněn použitím stránkování dat. Data bych ukládal ve stejném formátu jako nyní, pouze by byla rozdělena do bloků s určitým počtem rámců. Do informací o záznamu bych následně uložil polohu každého bloku uvnitř souboru s binárními daty. Při zobrazení rámců v tabulce se ze souboru podle indexů načtou pouze data ze dvou sousedících bloků. Při procházení tabulky by se následně provádělo obnovení obsahu stránek podle indexů požadovaných rámců.

Tímto postupem bude v operační paměti načteno vždy jen omezené množství dat. Předem však není možné říci, jaký vliv bude mít popsaný způsob načítání dat na rychlost obnovování obsahu tabulky. Lze odhadnout, že rychlost zobrazení dat bude hodně záviset na velikosti bloků a počtu používaných stránek.

Komunikace pomocí dvou soketů - V současné době jsou binární data serverem zakódována algoritmem Base64 a předávána uvnitř XML zprávy. Zmíněný postup se ukázal jako velmi omezující. Když server odesílá krátké řetězce zakódovaných dat, tak XML Parser v klientské aplikaci generuje velké množství událostí zahlcujících operační systém.

Při použití druhého komunikačního soketu určeného pouze pro předávání binárních dat v nezakódované podobě by měl být tento jev zcela odstraněn. Zpracování dat na straně klientské aplikace tím bude značně zjednodušeno a urychleno.

Pokročilejší filtrace dat - Je vhodné rozšířit filtrační rozhraní o možnost předdefinování několika filtrů. Takto definované filtry by mohly být následně kombinovány.

Vyhledávání v datech - V programu by měla být doplněna funkce pro vyhledávání rámců podle zadaného kritéria. Značně to ulehčuje orientaci ve velkém množství dat.

Příloha A

HW a SW požadavky

V této příloze uvádím doporučený hardware a software pro překlad a běh aplikace. Všechna doporučení jsou určena pro verzi programu 0.5. U budoucích verzí se mohou požadované prostředky změnit.

A.1 Doporučený hardware

CPU – pro online čtení dat ze serveru postačí běžný jednojádrový procesor s taktem 2 GHz a více. Pro optimální běh aplikace je však vhodnější dvoujádrový procesor, který není tolik vytížen při zpracování dat ze serveru.

Operační paměť – zpracování 10-ti milionů rámců vyžaduje, aby počítač obsahoval minimálně 1,5 GB operační paměti při použití Microsoft Windows 2000 nebo XP. Při použití systému Windows Vista doporučuji minimálně 2 GB operační paměti.

Grafická karta – bezproblémový běh programu je zaručen i s grafickou kartou určenou pro kancelářské aplikace.

Pevný disk – samotná aplikace s konfiguračním souborem a všemi knihovnami má velikost cca 4 MB. Pro ukládání záznamů však doporučuji minimálně 500 MB volného místa na disku.

A.2 Doporučený software

A.2.1 SW pro spuštění aplikace

Operační systém – pro bezproblémový běh aplikace doporučuji některý z operačních systémů Microsoft Windows 2000 nebo XP. Lze použít i Microsoft Windows Vista, ale tento systém má větší nároky na hardware. Hlavně na grafickou kartu a operační paměť.

Platforma .NET Framework – aplikaci lze spustit pouze pokud je v operačním systému nainstalováno běhové prostředí .NET Framework verze 2.0.

A.2.2 SW pro překlad aplikace

Pro překlad zdrojových kódů programu je kromě předchozích prostředků nutná instalace balíčku Microsoft .NET Framework SDK. Tento balík obsahuje podpůrné programy včetně Debuggeru a nástroje pro jazykovou lokalizaci formulářů. Po instalaci SDK je umožněn pouze překlad z příkazové řádky, který není příliš pohodlný.

Vhodnější je úprava a překlad zdrojových kódů aplikace ve vývojovém prostředí Microsoft Visual Studio verze 2005. V současné době se na trhu objevila i verze 2008, kterou jsem zatím neměl možnost vyzkoušet, a proto ji zatím nedoporučuji.

Příloha B

DTD soubory komunikačního protokolu

B.1 DTD komunikace od klienta k serveru.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT pbanalyzer (pbclient, pbconfig, pbdata, pbdatastop)>
<!ELEMENT pbclient EMPTY>
<!ATTLIST pbclient
          ver CDATA #REQUIRED
          id CDATA #REQUIRED>
<!ELEMENT pbconfig (pbset+, pbget+)>
<!ELEMENT pbset EMPTY>
<!ATTLIST pbset
          baudrate CDATA #IMPLIED
          tsreset CDATA #IMPLIED
          filteron CDATA #IMPLIED
          triggeron CDATA #IMPLIED>
<!ELEMENT pbget EMPTY>
<!ATTLIST pbget
          baudrate CDATA #IMPLIED
          tsreset CDATA #IMPLIED
          filteron CDATA #IMPLIED
          triggeron CDATA #IMPLIED>
<!ELEMENT pbdata EMPTY>
<!ATTLIST pbdata
          count CDATA #REQUIRED>
<!ELEMENT pbdatastop EMPTY>
```

B.2 DTD komunikace od serveru ke klientovi

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT pbanalyzer (pbserver, pbconfig, pbdata)>
<!ELEMENT pbserver EMPTY>
<!ATTLIST pbserver
          ver CDATA #REQUIRED
          id CDATA #REQUIRED>
<!ELEMENT pbconfig (pbans+)>
<!ELEMENT pbans EMPTY>
<!ATTLIST pbans
          baudrate CDATA #IMPLIED
          tsreset CDATA #IMPLIED
          filteron CDATA #IMPLIED
          triggeron CDATA #IMPLIED>
<!ELEMENT pbdata (d+)>
<!ATTLIST pbdata
          time CDATA #REQUIRED>
<!ELEMENT d (#PCDATA)>
```

Příloha C

Obsah přiloženého CD

K této práci je přiloženo CD s následujícím obsahem adresářů:

bin - obsahuje přeložené knihovny, spouštěcí soubor programu a XML souboru s nastavením.

data - obsahuje soubory uložených testovacích záznamů.

doc - obsahuje soubor s textem diplomové práce dp_2008_kautsky_ales.pdf.

- **latex** - obsahuje všechny obrázky a zdrojové kódy textu napsaného v LaTeXu.

dtd - obsahuje DTD soubory s definicí elementů komunikačního protokolu.

src - obsahuje soubor řešení (PB Remote Analyzer.sln) a podadresáře se zdrojovými kódy jednotlivých projektů, ze kterých se aplikace skládá.

- **Communication** - projekt se třídami pro obsluhu komunikace se serverem.
- **DataProcesing** - projekt se třídami pro zpracování dat.
- **Dokumentace** - dokumentace vygenerovaná programem Sandcastle z XML komentářů obsažených v souborech se zdrojovými kódy.
- **NewControls** - projekt s novými a upravenými komponentami pro grafické rozhraní aplikace.
- **PB Remote Analyzer** - hlavní projekt aplikace, který obsahuje vstupní bod programu a zdrojové kódy všech formulářů.
- **Settings** - projekt se třídami pro správu a ukládání nastavení.

test_server - obsahuje testovací serverovou aplikaci napsanou v .NETu.

Seznam obrázků

1.1 Blokové schéma analyzátoru 2

2.1 Porovnání s ISO/OSI modelem 3
2.2 Struktura UART znaku 6
2.3 Rámec pevné délky bez datové sekce 6
2.4 Rámec s proměnnou délkou datové sekce 6
2.5 Rámec s pevnou délkou datové sekce 7
2.6 Rámec pro předávání tokenu 7
2.7 Potvrzovací rámec (*Short Acknowledge*) 7
2.8 Formát adresy a rozšířené adresy 8
2.9 Graf závislosti délky *Message Cyklu* na přenosové rychlosti. 13
2.10 Graf závislosti počtu MC za sekundu na přenosové rychlosti. 14
2.11 Graf závislosti t_{SR} na přenosové rychlosti pro 30 stanic typu slave. 16
2.12 Graf závislosti t_{RR} na počtu masterů a přenosové rychlosti. 18

3.1 Blokové schéma platformy .NET 20
3.2 Blokové schéma aplikace 24

5.1 Základní struktura hlavního okna aplikace 32
5.2 Výsledné okno aplikace 33
5.3 Porovnání režimů vykreslování položek u `TreeView` 33
5.4 Formulář s nastavením aplikace 38

6.1 Formulář `ComConfigForm` pro nastavení spojení a serveru 41
6.2 Schéma navázání spojení mezi klientem a serverem 42
6.3 Schéma vícevláknového modelu pro obsluhu komunikace 48
6.4 Stavový diagram komunikace na straně klientské aplikace 52
6.5 Formát dat na serveru 53

6.6 Ilustrace kódování Base64 55

7.1 Datový panel hlavního okna aplikace 62
7.2 Obsah panelů s doplňkovými informacemi rámce 63
7.3 Formulář pro nastavení filtru zobrazovaných dat 64
7.4 Formulář pro zobrazení statistik 67

Seznam tabulek

2.1 Identifikace typů zpráv podle SAPů a FDL služeb. 10
2.2 Závislost t_{MC} [ms] na přenosové rychlosti a velikosti dat. 12
2.3 Závislost R_{SYS} [N/s] na přenosové rychlosti a velikosti dat. 14
2.4 Závislost t_{SR} [ms] na přenosové rychlosti a velikosti dat. 15
2.5 Závislost t_{RR} [ms] na počtu stanic master a přenosové rychlosti. 17

4.1 Stromová struktura tříd s nastavením 26

6.1 Definice elementů komunikačního protokolu a jejich význam 45
6.2 Popis vybraných vlastností a metod třídy `XmlTextReader` 50
6.3 Délka časové značky podle CF. 53

Seznam použitých zkratek

t_{bit}	*Bit Time* - převrácená hodnota přenosové rychlosti sběrnice Profibus. Udává za jak dlouho je přenesen jeden bit
API	*Application Programming Interface* - systémové rozhraní pro programování aplikací.
BCL	*Base Class Library* - knihovna základních tříd platformy .NET Framework.
CF	*Control Field* - informační pole uvnitř rámce udávající délku časové značky.
CLR	*Common Language Runtime* - běhové prostředí platformy .NET Framework.
CTS	*Common Type System* - společný systém datových typů pro programovací jazyky platformy .NET Framework.
DA	*Destination Address* - adresa cílové stanice.
DDLM	*Direct Data Link Mapper* - blok, který se stará o mapování DP funkcí na DP zprávy a naopak.
DNS	*Domain Name Server* - server překládající IP adresu na jméno počítače a naopak.
DP	*Decentralized Peripherals* - nejrozšířenější protokol sběrnice Profibus.
DTD	*Document Type Definition* - definice elementů XML dokumentu.
ED	*End Delimiter* - ukončovací znak FDL rámce.
FCS	*Frame Checksum* - kontrolní součet FDL rámce.
FMS	*Fieldbus Message Specification* - nejstarší protokol sběrnice Profibus.
GC	*Garbage Collector* - nedílná součást platformy .NET Framework, která se stará o správu paměti.
GUI	*Graphical User Interface* - grafické uživatelské rozhraní.
IP	*Internet Protokol* - komunikační protokol určený pro Internet.
ISO/OSI	Referenční model pro standardizaci počítačových sítí.

LAN	*Local Area Network* - lokální počítačová síť.
LAS	*List of Active Stations* - seznam aktivních stanic na sběrnici Profibus.
LF	*Length Field* - informační pole uvnitř rámce udávající délku FDL rámce.
MAC	*Medium Acces Control* - blok, který řeší přístup na fyzické médium sítě.
MSDN	*Microsoft Developer Network* - webové stránky společnosti Microsoft určené pro podporu vývojářů.
MSIL	*Microsoft Intermediate Language* - objektový jazyk platformy .NET Framework.
OS	Zkratka operačního systému.
PLC	*Programmable Logic Controller* - programovatelný logický automat
RTF	*Rich Text Format* - jednoduchý značkovací jazyk pro formátování textu.
SA	*Source Address* - adresa zdrojové stanice.
SAP	*Service Acces Point* - přístupový bod služby.
SAX	*Simple API for XML* - rozhraní pro zpracování XML z proudu dat.
SD	*Start Delimiter* - bajt označující začátek FDL rámce.
SDA	*Send Data with Acknowledge* - FDL služba.
SDN	*Send Data No Acknowledge* - FDL služba.
SRD	*Send and Request Data* - FDL služba.
TCP	*Transmision Control Protocol* - protokol transportní vrstvy komunikačního systému.
TS	*Time Stamp* - časová značka rámce.
UART	*Universal Asynchronous Receiver/Transmitter* - univerzální asynchr. přijímač/vysílač
VB	*Visual Basic* - programovací jazyk.
VS	*Visual Studio* - vývojové prostředí určené pro platformu .NET Framework.
XML	*Extensible Markup Language* - rozšířitelný značkovací jazyk.

Literatura

[1] SAMEK, Martin, *Server analyzátoru protokolu Profibus DP.* ČVUT v Praze, Elektrotechnická fakulta, Katedra řídicí techniky, 2008. Diplomová práce.

[2] *PROFIBUS Specification: Normative Parts of PROFIBUS -FMS, -DP, -PA according to the European Standard EN 50 170 Volume 2*, vol. 1.0. PROFIBUS International, 1998. http://www.profibus.com.

[3] TRNKA, Pavel, *Profibus DP Master na PC.* ČVUT v Praze, Elektrotechnická fakulta, Katedra řídicí techniky, 2004. Diplomová práce.

[4] BLEPTA, Jan, *Diagnostický uzel pro Profibus.* ČVUT v Praze, Elektrotechnická fakulta, Katedra řídicí techniky, 2001. Diplomová práce.

[5] Microsoft, *.NET Framework Home.* [online]. http://msdn2.microsoft.com/en-us/netframework/default.aspx.

[6] ROBINSON, Simon; at al, *C# Programujeme profesionálně*, vol. 1. Brno: Computer Press, 2003. ISBN 80-251-0085-5.

[7] GUNNERSON, Eric, *Začínáme programovat v C#.* Brno: Computer Press, 2001. ISBN 80-7226-525-3.

[8] Wikipedia, *Windows API.* [online]. http://en.wikipedia.org/wiki/Windows_API.

[9] *What is Mono.* [online]. http://www.mono-project.com/Main_Page.

[10] *DotGNU Project.* [online]. http://www.dotgnu.org/.

[11] *Extensible Markup Language (XML).* [online]. http://www.w3.org/XML/.

[12] Kosek, Jiří, *Základy jazyka XML.* [online]. http://www.kosek.cz/clanky/swn-xml/syntaxe.html.

[13] MACDONALD, Matthew, *Pro .NET 2.0 Windows Forms and Custom Controls in C#*. Berkeley: Apress, 2006. ISBN 1-59059-439-8.

[14] DOSTÁLEK, Libor; KABELOVÁ, Alena, *Velký průvodce protokoly TCP/IP a systémem DNS*. Praha: Computer Press, 2000. ISBN 80-7226-323-4.

[15] Wikipedia, *Base64*. [online]. http://en.wikipedia.org/wiki/Base64.

[16] Microsoft, *Rich Text Format (RTF) Specification*. [online]. http://latex2rtf.sourceforge.net/rtfspec.html.