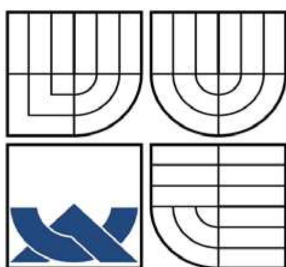
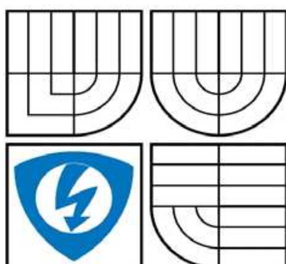


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF TELECOMMUNICATIONS

Rozšíření komunikačního protokolu MPKT na platformě .NET

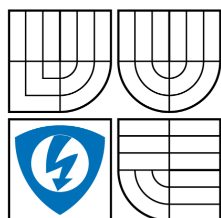
MPKT communication protocol extension on .NET platform

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. Tomáš Görlich
AUTHOR

VEDOUCÍ PRÁCE Ing. Jan Jeřábek
SUPERVISOR

BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav telekomunikací

Diplomová práce

magisterský navazující studijní obor

Telekomunikační a informační technika

Student: Bc. Tomáš Görlich ***ID:*** 83783

Ročník: 2 ***Akademický rok:*** 2008/2009

NÁZEV TÉMATU:

Rozšíření komunikačního protokolu MPKT na platformě .NET

POKYNY PRO VYPRACOVÁNÍ:

Proveďte analýzu dosavadní verze komunikačního protokolu MPKT. Nastiňte možné varianty jeho rozšíření, vylepšení a zefektivnění. Vyhodnoťte, který z programovacích jazyku, dostupných na platformě .NET, bude pro následnou implementaci nejvhodnější. Zohledněte zejména hledisko využití pokročilejších funkcí jazyka určených pro práci v sítovém prostředí. V rámci diplomové práce vytvořte referenční řešení, včetně patřičné dokumentace a doplnění výukových textu.

DOPORUČENÁ LITERATURA:

[1] Troelsen, A.: Pro C# and the .NET 3.5 Platform, Fourth Edition. Apress, Berkeley, USA, 2007, 1370 stran.
[2] Malý, J.: Implementace komunikačního protokolu MPKT v C++, VUT v Brně, 2008.

Termín zadání: 9.2.2009 ***Termín odevzdání:*** 26.5.2009

Vedoucí práce: Ing. Jan Jeřábek

prof. Ing. Kamil Vrba, CSc.
Předseda oborové rady

UPOZORNĚNÍ:

Autor diplomové práce nesmí při vytváření diplomové práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následku porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledku vyplývajících z ustanovení § 152 trestního zákona č. 40/1961 Sb.

Anotace

Tato diplomová práce je zaměřena na rozšíření komunikačního protokolu MPKT 2008. Byla provedena analýza této verze protokolu a byly vzneseny návrhy na vylepšení a rozšíření této verze o další možnosti. Na základě tohoto návrhu byla vytvořena nová verze protokolu. Dále byla představena platforma .NET a popsána její architektura. Pro implementaci nové verze protokolu MPKT byl vybrán programovací jazyk C# dostupný na platformě .NET. Byly představeny prostředky tohoto programovacího jazyka pro práci v síťovém prostředí a práci s vlákny. Bylo vysloveno zhodnocení o vhodnosti použití tohoto programovacího jazyka v praktické výuce. Nakonec byl proveden návrh aktualizace výukových textů do předmětu Pokročilé komunikační techniky.

Klíčová slova:

komunikační protokol, platforma .NET, C#

Abstract

This thesis is focused on extension of MPKT 2008 communication protocol. It has been made analysis of this protocol version and proposals to improve and extend this version with other options have been raised. It has been created a new version of the protocol on the basis of the proposals. Then the .NET platform has been presented and its architecture has been described. The C# language available on the .NET platform has been chosen to implement the new protocol version and its facilities for network communication and threading have been described. It has been said an assessment of the application of this programming language in practical lessons. A proposal to update tutorial texts has been made.

Keywords:

communication protocol, .NET platform, C#

GÖRLICH, T. *Rozšíření komunikačního protokolu MPKT na platformě .NET*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2009. 68 s. Vedoucí diplomové práce Ing. Jan Jeřábek.

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci na téma Rozšíření komunikačního protokolu MPKT na platformě .NET vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením tohoto projektu jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení § 152 trestního zákona č. 140/1961 Sb.

V Brně dne ....

podpis autora

Obsah

1 Základní pojmy síťové komunikace ... 8

2 Seznámení s projektem MPKT verze 2008 ... 10

2.1 Protokol MPKT ... 10
2.1.1 Kontrola při přijetí paketu ... 10
2.1.2 Klient – popis funkce ... 11
2.1.3 Server – popis funkce ... 12

3 Analýza projektu MPKT verze 2008 ... 14

3.1 Návrhy vylepšení projektu MPKT ... 14

4 Platforma .NET ... 16

4.1 Struktura .NET Frameworku ... 16
4.1.1 CLR – Common Language Runtime ... 17
4.1.2 FCL – Framework Class Library ... 18
4.1.3 CIL – Common Intermediate Language ... 18
4.1.4 Assemblies ... 19
4.1.5 ADO.NET: Data a XML ... 19
4.1.6 Webové služby ... 19
4.1.7 Windows formuláře ... 20

4.2 .NET Remoting ... 20

4.3 .NET a projekt Mono ... 21

5 Porovnání jazyků C++ a C# ... 22

5.1 Jazyk C# ... 22

5.2 Síťové prostředky v jazyce C# ... 23
5.2.1 System.Net.Sockets ... 23

5.3 Vlákna v jazyce C# ... 27

6 Projekt MPKT na platformě .NET ... 29

6.1 Popis nového projektu ... 29
6.1.1 Třída Paket ... 29
6.1.2 Třída Protokol ... 30

6.2 Server – popis funkce ... 34

6.3 Klient – popis funkce ... 40

7 Doplňkové studie k vypracování programu ... 46

7.1 Převod struktury/třídy na proud bajtů ... 46

7.2 Dynamický seznam klientů ... 48

7.3 Zašifrování přenášených zpráv ... 48

Závěr ... 50

Literatura ... 51

Seznam příloh ... 52

Úvod

Cílem diplomové práce je provedení analýzy dosavadní verze komunikačního protokolu MPKT verze 2008, kterým se v rámci praktických cvičení v předmětu Pokročilé komunikační techniky demonstrují základní metody práce s komunikačními protokoly. Výsledkem analýzy bude nastínění možných variant rozšíření, vylepšení a zefektivnění protokolu, a tím bude dán základ pro vytvoření nové softwarové aplikace.

V první části této práce je nejprve popsána původní verze komunikačního protokolu MPKT. Poté následuje teoretická rozprava o v současné době stále rozšířenější platformě .NET. Vylepšený komunikační protokol MPKT je cílený právě na tuto platformu. Dále je představen programovací jazyk C#, dostupný na platformě .NET, který byl zvolen pro implementaci tohoto protokolu. Praktická část této práce je založená na vytvoření klientské a serverové aplikace včetně uživatelského rozhraní, které spolu budou komunikovat prostřednictvím rozšířeného protokolu. Cílem je vytvořit referenční řešení včetně patřičné dokumentace, které zohlední zejména pokročilejší možnosti komunikace na úrovni socketů a princip vláken v jazyku C#. Dále je proveden návrh aktualizace výukových textů do předmětu Pokročilé komunikační techniky. V upravených textech je představen jazyk C# od základních principů objektově orientovaného programování až po jeho možnosti pro práci v síťovém prostředí na úrovni socketů a práci s vlákny.

1 Základní pojmy síťové komunikace

V této kapitole jsou uvedeny základní pojmy a prostředky síťové komunikace. Čerpáno z literatury [1].

TCP/IP

TCP/IP představuje sadu protokolů pro komunikaci v počítačové síti a je hlavním protokolem celosvětové sítě internet. Komunikační protokol představuje množinu pravidel, které určují syntaxi a význam jednotlivých zpráv při komunikaci. Vzhledem ke své složitosti je síťová komunikace rozdělena do tzv. vrstev, které znázorňují hierarchii činností. Výměna informací mezi vrstvami je přesně definována. Každá vrstva využívá služeb vrstvy nižší a poskytuje své služby vrstvě vyšší.

Architektura TCP/IP je členěna do čtyř vrstev:

- *aplikační vrstva* - zahrnuje uživatelské aplikace, které generují data pro přenos a zpracovávají přijatá data,
- *transportní vrstva* - realizuje komunikaci koncových uzlů, multiplexuje a demultiplexuje datový tok různých aplikací v koncovém uzlu, může zajistit spolehlivý přenos s detekcí ztracených jednotek,
- *síťová vrstva* - definuje adresní schéma tak, aby pomocí ní bylo možné adresovat každý uzel v síti, na základě těchto adres provádí vyhledávání cesty od vysílače k přijímači, tzv. směrování,
- *vrstva síťového rozhraní* - umožňuje přístup k fyzickému přenosovému médiu, je specifická pro každou síť v závislosti na její implementaci.

Protokol TCP

Je protokol transportní vrstvy TCP/IP modelu. Jedná se o spojově orientovaný protokol, což znamená, že před samotnou komunikací se mezi koncovými prvky nejprve naváže spojení. Všechna odeslaná data se potvrzují a nakonec je nutné spojení uzavřít. TCP segment obsahuje hlavičku a data. Součástí hlavičky TCP segmentu je tzv. port, což je dva bajty dlouhé číslo, které náleží aplikační vrstvě.

Protokol UDP

Je v podstatě alternativou k protokolu TCP, avšak s tím rozdílem, že se jedná o nespojovanou a nepotvrzovanou službu. Slouží k práci s přenosy dat bez jakékoliv kontroly nebo potvrzování. Je vhodný zejména v situacích, kdy by spojení pomocí TCP bylo velkou zátěží pro síť.

Protokol IP

Je protokol síťové vrstvy TCP/IP modelu používaný pro přenos dat přes paketové sítě. Tvoří základní protokol dnešního internetu. Komunikace mezi koncovými uzly v síti, které jsou jednoznačně identifikovány IP adresou, probíhá předáváním tzv. IP paketů. Poskytuje nespolehlivou datagramovou službu.

IP adresa

Udává jednoznačnou identifikaci síťového rozhraní v síti.

- IPv4 - 32 bitové číslo, zapisované po jednotlivých bajtech, oddělenými tečkami, v současnosti je již znatelný nedostatek IPv4 adres.
- IPv6 – 128 bitové číslo, počet možných adres je 2^{128}, zapisuje se jako skupina osmi hexadecimálních čísel, např. 2001:0718:1c01:0016:0214:22ff:fec9:0ca5.

V rámci protokolu IPv4 se k IP adrese váže maska podsítě, rovněž čtyřbajtový údaj, který udává, která část IP adresy je adresa podsítě a která část adresa konkrétní stanice.

Socket

Socket je prostředek, který otevírá možnost komunikace na konkrétním portu určitého síťového rozhraní. Vytváří se na aplikační vrstvě a jeho obsluha je věcí operačního sytému. V zásadě je se socketem možno pracovat v jednom ze dvou režimů:

- blokovací – zastavení programu během čekání na příchozí požadavek a znemožnění obsloužení více požadavků najednou, což nebývá vždy žádoucí,
- neblokovací – umožňuje obsloužení více příchozích požadavků nebo umožňuje programu vykonávat jinou činnost při čekání na příchozí požadavek.

Alternativní metodou k neblokovacímu režimu je vyžití techniky vláken při běhu programu

Technika vláken

Princip vícevláknové práce je moderní způsob správy programu v oddělených modulech – vláknech, které operační systém provozuje současně a střídá jejich činnost neustálým přepínáním. Vlákna se chovají jako samostatné programy a je možné jim předat parametr, který má význam ve funkci rozlišení jednotlivých vláken mezi sebou. V rámci komunikace po síti může být jako parametr vlákna předán např. socket, pokud se zakládá pro každý obsloužený požadavek na straně serveru vlastní vlákno, což je již zmíněná alternativa k neblokovacímu režimu socketu.

2 Seznámení s projektem MPKT verze 2008

Cílem projektu v předmětu Pokročilé komunikační techniky (MPKT) je osvojit si základní metody práce s komunikačními protokoly v jazyce C/C++ na platformě Windows. Projekt tvoří dva samostatné programy, které spolu komunikují jako server a klient. Komunikující strany využívají spojovaného protokolu TCP na portu 5000, přičemž server je schopen s využitím techniky vláken obsloužit více než jednoho klienta. Dále server podporuje i komunikaci prostřednictvím protokolu UDP na portu 6000, kdy na speciální zprávu obdrženou od klienta odešle server tuto zprávu všesměrově všem serverům v rámci lokální sítě. Čerpáno z literatury [1].

2.1 Protokol MPKT

Komunikační protokol MPKT obsahuje vlastní hlavičku, která zabraňuje tomu, aby se serverem komunikovala jakákoliv jiná aplikace na bázi protokolu TCP. Hlavička obsahuje čtyři základní údaje, a to identifikační číslo klienta, délku odesílané zprávy, typ operace a kontrolní součet zprávy, bude-li v paketu přítomna. Délka hlavičky paketu 6 bajtů a zpráva má povolenou délku 0 – 1000 bajtů, viz obrázek 2.1.

číslo klienta 1B	délka zprávy 2B	operace 1B	CRC16 2B	0-1000B
hlavička				zpráva

Obr. 2.1: Paket MPKT 2008

Vysvětlení všech položek záhlaví paketu MPKT:

Číslo klienta – každý klient se identifikuje vlastním unikátním číslem v rozsahu 1 – 255 (1B). Pokud by se klient přihlásil s číslem již požívaným jiným klientem, server komunikaci odmítne.

Délka zprávy – v případě textové komunikace se do tohoto pole vypíše počet bajtů odesílané zprávy, jinak se zde ponechá nula.

Operace – protokol pracuje se šesti druhy zpráv, které se identifikují pomocí písmen. Jde o jednobajtovou informaci datového typu char. Jednotlivé typy zasílaných zpráv obsahuje tabulka 2.1.

2.1.1 Kontrola při přijetí paketu

Po přijetí paketu se provede kontrola hlavičky, aby se zabránilo případné komunikaci s jiným programem. Kontrola se provádí v několika úrovních:

- operace – každá zpráva musí tuto položku obsahovat a musí být z výše uvedené množiny **C**, **M**, **I**, **D**, **A**, **T,** jinak server s daným klientem komunikaci ukončí,
- číslo klienta – pokud zpráva není typu **D**, číslo klienta musí odpovídat klientu, se kterým se komunikace právě vede. Na serveru se číslo klienta ukládá do dynamického seznamu,

- délka přijatých dat – pokud zpráva je typu **C** nebo **M**, je délka přijatých dat rovna políčku délka + 6. Pokud je jiného typu, je délka vždy 6. V případě nesouhlasné hodnotě funkce `recv()` se komunikace ukončí,
- CRC16 – jde-li o textovou zprávu, provádí se kontrola přijaté zprávy pomocí algoritmu CRC16.

Tímto způsobem se dosáhne nadstandardního zabezpečení přenášených dat. Pokud je detekována jakákoliv chyba při kontrole hlavičky příchozího paketu, uzavře se spojení a komunikace serveru s daným klientem se ukončí s chybou.

Tab. 2.1: Typy zasílaných zpráv mezi klientem a serverem

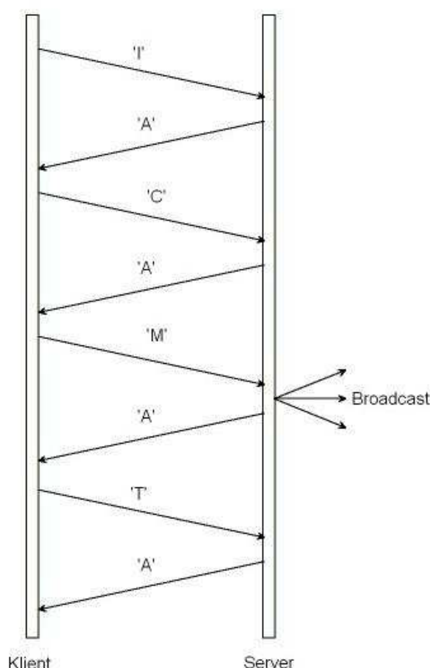
C	textová zpráva	Zprávu zasílá klient serveru, který ji klientu potvrdí a zprávu zobrazí. Je zahrnuta kontrola CRC a zaplněno políčko délka zprávy.
M	hromadná textová zpráva	Zprávu zasílá klient serveru, který ji klientu potvrdí a zprávu zobrazí a poté tuto zprávu odešle pomocí protokolu UDP všem ostatním serverům v rámci lokální sítě. Je zahrnuta kontrola CRC a zaplněno políčko délka zprávy.
I	inicializace komunikace	Zprávou se inicializuje komunikační protokol po zahájení TCP komunikace mezi oběma stranami. Server zjistí, zda je požadované číslo klienta unikátní a uloží si ho do dynamického seznamu. CRC ani délka zprávy se neuplatní.
D	zamítnutí	Zprávou server oznamuje klientovi zamítnutí komunikace z důvodu duplicity čísla s klienta a následuje ukončení komunikace s tímto klientem. CRC ani délka zprávy se neuplatní.
A	potvrzení	Kdykoliv je-li potřeba potvrzení, posílá server tento typ zprávy. CRC ani délka zprávy se neuplatní.
T	ukončení	Zprávou oznamuje klient serveru, že komunikaci končí, a očekává potvrzení od serveru. CRC ani délka zprávy se neuplatní.

2.1.2 Klient – popis funkce

Klient se spouští v příkazové řádce a jsou mu být předány dva parametry: IP adresa serveru a identifikační číslo klienta. Ihned poté se provede inicializace komunikace se serverem pomocí zprávy typu **I**. V případě jakýchkoliv nastanuvších chyb se klient ukončí s chybovým hlášením, a to platí po celou dobu komunikace. Po úspěšné inicializaci klient začne načítat uživatelem zadávaný text ze standardního vstupu a posílá jej ve výše popsaném paketu na server – zpráva typu **C**. V případě, že prvním zadaným znakem zprávy je *, dojde k odeslání zprávy typu **M**. Server potvrdí klientovi správné přijetí a pomocí protokolu UDP zprávu dále

přepošle všem serverům v rámci lokální sítě. Při těchto dvou typech zpráv musí být v hlavičce paketu vyplněny položky délka zprávy a CRC16.

Pokud uživatel zadá řetězec “**exit**“, klient tuto zprávu neodesílá, ale zpracuje ji jako pokyn k ukončení komunikace. Klient tak odešle zprávu typu **T** a nechá si ji od serveru potvrdit. Poté dojde k ukončení spojení se serverem a k ukončení klientské aplikace.



Obr. 2.2: Schéma komunikace s protokolem MPKT 2008

2.1.3 Server – popis funkce

Server se také spouští v příkazové řádce a nepředává se mu žádný parametr. Server si pojmenuje TCP socket na portu 5000 a UDP socket na portu 6000 a zahájí na nich naslouchání. Pro každého klienta je spuštěno samostatné vlákno pro vedení komunikace přes protokol TCP.

Při požadavku na TCP spojení se pracuje se standardním založením spojení v jazyce C++, které se provádí pomocí standardních funkcí `listen()`, `bind()` a `accept()`. Při akceptování nového spojení se založí nové vlákno, kterému je jako parametr předán nově vzniklý socket, na kterém proběhne komunikace s daným klientem.
Další postup je následující:

1. očekávání zprávy typu **I** k inicializaci. Přijde-li jiný typ zprávy, spojení s tímto klientem se ukončí.
2. ověření, zda se klient s uvedeným číslem již nenachází v aktuálním seznamu klientů. Pokud ne, odešle se přijetí spojení pomocí zprávy typu **A**, jinak **D** následováno ukončením komunikace s tímto klientem.

3. probíhající komunikace pomocí zpráv tří typů:
   - **C** - vypíše se zpráva na obrazovku s číslem klienta a odešle se potvrzení **A**,
   - **M** - vypíše se zpráva na obrazovku s číslem klienta a odešle se pomocí UDP na portu 6000 všesměrově ostatním serverům v rámci lokální sítě,
   - **T** - server žádost ukončení komunikace klientovi potvrdí a ukončí komunikaci s tímto klientem. Je nutné též odstranit číslo tohoto klienta ze seznamu aktuálně připojených klientů.

Server se neukončí, dokud se nepoužije sekvence kláves Ctrl+C.

3 Analýza projektu MPKT verze 2008

Projektem MPKT se demonstrovaly základní metody práce s komunikačními protokoly v programovacím jazyce C/C++ na platformě Windows. Samotný protokol MPKT byl navržen pro spolehlivou komunikaci mezi klientskou a serverovou aplikací na bázi protokolů TCP/IP.

V projektu MPKT se využívá tzv. blokovací režim socketů. To znamená, že se program při volání funkcí jako `accept()`, `recv()`a `recvfrom()` zastaví a čeká na příchozí požadavky. V případě klientské aplikace s blokovacím režimem vystačíme, protože postup komunikace mezi klientem a serverem je přesně dán pomocí očekávaných typů zpráv. Klient se připojí ke vzdálenému serveru a očekává kladné potvrzení o přijetí spojení. Poté již klient může odesílat textové zprávy serveru a po každé odeslané zprávě očekává od serveru potvrzení o správnosti přijaté zprávy. Na straně serverové aplikace bychom však s blokovacím režimem socketů již nevystačili, protože server musí obsloužit více klientů současně. V tomto případě byl však namísto neblokovacího režimu socketů využit moderní způsob správy programu v oddělených modulech – vláknech, které operační systém provozuje současně, a provádění programu jednotlivými vlákny neustále střídá. Pro využití vláken v systému Windows je v jazyce C++ k dispozici třída `TTHread`. Pomocí vláken se tedy zajistí, že server naslouchá současně na dvou portech, konkrétně TCP a UDP portech, a po přijetí spojení na žádost klienta tomuto klientu vytvoří nové vlákno pro komunikaci pomocí protokolu TCP. Po ukončení komunikace s tímto klientem je vlákno zrušeno, stejně tak jsou uzavřeny síťové prostředky této relace.

Přenesené textové zprávy jsou na straně příjemce zkontrolovány, zda v nich během přenosu nedošlo k chybě. Je provedena kontrola CRC (kontrolní redundantní součet) příchozí zprávy, který je porovnán s CRC obsaženým v hlavičce příchozího paketu. Také je provedena kontrola délky příchozí zprávy s délkou obsaženou rovněž v hlavičce příchozího paketu. Takový způsob kontroly bezchybného přenosu je jistě dostačující.

3.1 Návrhy vylepšení projektu MPKT

- **Komfortnější uživatelské rozhraní**

Jak server tak i klient jsou původně pouze konzolové aplikace. Pro pohodlnější zobrazování příchozích zpráv a informací o navázaných spojeních bude vhodné oběma aplikacím přidat formulářové uživatelské rozhraní s intuitivním ovládáním.

- **Využití časovačů**

Při čekání klientské aplikace na jakékoli potvrzení ze strany serveru (žádost o spojení, žádost o ukončení spojení, potvrzení přijetí zprávy, atd.) by bylo vhodné zavést časovač. Pokud by do vypršení doby časovače nepřišlo očekávané potvrzení, byl by o tom klient informován, avšak komunikace se neukončí a obsluha klienta by měla předchozí operaci zopakovat, tj. znovu odeslat textovou zprávu, případně žádost o ukončení spojení. Server by po určité době

nečinnosti klienta měl toto spojení ukončit a uvolnit tak své prostředky pro požadavky dalších klientů.

- **Možnost psaní a odesílaní zpráv ze serverové aplikace**

Ve své základní podobě server pouze přijímá a zobrazuje textové zprávy zasílané svými klienty, popřípadě zobrazuje zprávy přicházející přes UDP port od jiných serverů. Bylo by však vhodné psát textové zprávy i na straně serveru a tyto zprávy odesílat svým klientům, viz následující návrh.
Zprávu ze serveru bude možno zaslat:

  - konkrétnímu klientovi,
  - všem připojeným klientům,
  - jen určité skupině připojených klientů.

- **Fragmentace textových zpráv**

Hlavička námi definovaného paketu se rozšíří a položky související s fragmentací obsahu textové zprávy. Podobně jako u protokolu IP bude možno textovou zprávu rozdělit do více fragmentů, bude-li délka zprávy přesahovat stanovenou mez a na straně příjemce bude z fragmentů opět složena. Délka fragmentu bude stanovena na maximálně 100B.

- **Využití zašifrované komunikace**

Za účelem zvýšeného zabezpečení přenášených textových zpráv mezi klientem a serverem by bylo vhodné obsah těchto zpráv zašifrovat. Pro zašifrování zprávy je možné vybrat si z nabídky šifrovacích algoritmů. Šifrovací algoritmy se z obecnějšího hlediska dělí na symetrické a asymetrické. Symetrické šifrovací algoritmy používají pro šifrování a dešifrování stejný klíč. Příkladem je algoritmus DES. Asymetrické algoritmy pracují s dvěma typy klíčů. Veřejný klíč slouží k zašifrování dat a není nutné ho skrývat. K dešifrování veřejným klíčem zašifrovaných dat se používá tajný soukromý klíč. Příkladem je algoritmus RSA. Server by si nejprve vygeneroval pár klíčů k šifrovacímu algoritmu RSA a svůj veřejný klíč by odeslal na stranu klienta. Klient si tento veřejný klíč uloží. Poté si klient vygeneruje tajný klíč pro symetrické šifrování textových zpráv. Tento tajný klíč klient zašifruje veřejným klíčem serveru a odešle mu jej. Tento návrh šifrované komunikace není ve vytvořené aplikaci implementován.

4 Platforma .NET

Platforma .NET byla představena firmou Microsoft v roce 2000. Tato platforma představuje novou generaci systému vývoje aplikací založeném na řízeném běhovém prostředí, obohaceném o rozsáhlou sadu základních tříd. Hlavními důvody vedoucí k více než čtyřletému vývoji této platformy byly:

- nekompatibilita jednotlivých programovacích jazyků a s tím související obtížná spolupráce mezi programy/knihovnami napsanými v odlišných jazycích (např. C++ a Visual Basic),
- vysoká chybovost aplikací (chyby v práci s pamětí, neplatné konverze datových typů),
- problémy s verzemi knihoven (obtížná práce s provozem více verzí knihoven),
- zastaralý a nepřehledný způsob vývoje dosavadních webových aplikací.

Všechny tyto problémy efektivně řeší platforma .NET, a to použitím řízeného běhového prostředí, systémem assemblies (viz oddíl 4.1.4), což jsou základní stavební prvky aplikací, a novou technologií ASP .NET pro vývoj webových aplikací.
Srdcem platformy je .NET Framework, jehož jádro se stará o tyto úkoly:

- je prostředím pro běh aplikací, tzn. že aplikace pro něj napsané jsou s ním pevně spjaty a bez něho je nelze spustit,
- obsahuje velmi širokou řadu knihoven, objektově orientovaných tříd usnadňujících řadu úkolů (bezpečnost, komunikaci, práci s databázemi a datovými zdroji, …),
- řeší rozšířený problém s přepisováním knihoven různých verzí tak, že zavádí verzování knihoven a digitální podpis knihoven,
- přináší možnost programování prakticky v kterémkoli programovacím jazyku s využitím patřičného kompilátor,
- zavádí stejný programový model pro jazyky Visual Basic .NET, C++ .NET, C# .NET, kdy knihovny napsané v jednom z těchto jazyků jsou bez konverzí použitelné i pro ostatní jazyky [3].

.NET Framework stojí jako nadstavba nad operačním systémem. Obsahuje runtimové běhové prostředí CLR (Common Language Runtime) a knihovny tříd nazvané FCL (Framework Class Library). FCL je vytvořena nad CLR a poskytuje služby vyžadované moderními aplikacemi [3].

4.1 Struktura .NET Frameworku

.NET Framework je sada technologií, které jsou integrovány do Microsoft .NET platformy. Poskytuje následující komponenty:

- CLR (Common Language Runtime) – běhové prostředí,
- FCL (Framework Class Library) – knihovna základních tříd,
- ADO.NET: Data a XML,

- Webové služby,
- Windows formuláře.

Bližší popis k jednotlivým vrstvám v podkapitolách 4.1.

Na obrázku 4.1. jsou zachyceny jednotlivé vrstvy .NET Frameworku.

Obr. 4.1: Struktura .NET Frameworku

Následuje podrobný popis vrstev .NET Frameworku.

4.1.1 CLR – Common Language Runtime

CLR je moderní prostředí běhu programu, které se stará o vykonávání uživatelského kódu a poskytuje určité služby, jako je kompilace JIT (Just In Time), správa paměti, správa výjimek, podpora ladění a profilování a správa integrovaného zabezpečení a oprávnění [2].

Tradiční překladače jsou zaměřeny na určitý procesor. Přijímají zdrojové soubory v určitém jazyku a vytvářejí binární soubory obsahující proudy instrukcí v nativním jazyku cílového procesoru. Tyto binární soubory lze následně vykonat přímo na cílovém procesoru. Kompilátor .NET Frameworku pracuje tak, že se nezaměřuje na konkrétní nativní procesor, ale přejímá zdrojové soubory a vytváří binární soubory obsahující zprostředkující reprezentaci zdrojových instrukcí, vyjádřených jako kombinace metadat a jazyka CIL (Common Intermediate Language). Aby bylo možné instrukce vykonat, musí se na koncovém počítači nacházet CLR. Metadata ve zkompilované softwarové komponentě činí danou komponentu samopopisnou. Jinak řečeno komponenty napsané v různých programovacích jazycích mohou přímo spolupracovat s jinými komponentami [2].

Generování kódu specifického k určitému procesoru se odkládá až na dobu spuštění programu a CLR má možnost vykonávat procesorově specifické optimalizace podle cílové architektury, na níž kód běží. Jak se bude vyvíjet architektura procesorů, postačí jen aktualizovat CLR.

4.1.2 FCL – Framework Class Library

Je to rozsáhlá síť tříd a dalších typů, jakými jsou rozhraní, struktury a výčty, které jsou rozděleny do více než 90 jmenných prostorů, poskytující základní systémové služby, jakými jsou:

- základní a rozšířené datové typy a zpracování chyb (jmenný prostor System),
- přístup k datům (jmenný prostor System.Data),
- prvky uživatelského rozhraní standardních aplikací pro Windows (jmenný prostor System.Windows.Forms),
- prvky uživatelského rozhraní webových aplikací (jmenný prostor System.Web.UI).

Například jazyky jako C#, Visual Basic .NET a JScript jsou implementované jako množiny tříd náležejících do odpovídajících oborů názvů, např. Microsoft.VisualBasic. Je možné vytvářet vlastní obory názvů pomocí příkazu `Namespace` [2].

4.1.3 CIL – Common Intermediate Language

Je výstupem každého kompilátoru na platformě .NET při generování řízeného kódu. Je to procesorově nezávislý jazyk podobající se assembleru. Jeho zavedení je snahou zvýšit přenositelnost programového kódu na různé druhy hardwarových zařízení. Jazyk CIL zachovává téměř všechny konstrukce původního zdrojového jazyka, ale tato reprezentace není na zdrojovém jazyku závislá, což dovoluje vývojářům vytvářet aplikace s využitím více různých programovacích jazyků, kdy pro konkrétní úkol si vyberou ten nejvhodnější jazyk a nemusejí se ani spoléhat na komponentové technologie, jakými jsou COM nebo CORBA [4], potřebují-li zamaskovat rozdíly mezi zdrojovými jazyky používanými k sestavení oddělených komponent nějaké aplikace [2].

Pro vykonání CIL kódu je potřeba jej přeložit do nativního strojového kódu za pomocí kompilátoru Just-in-time, zkráceně JITer.
Podle [3] se rozlišují 3 druhy Just-in-time kompilace:

- Kompilace v době instalace. Poskytuje výhodu v tom, že se CIL kód překládá pouze jednou, a pak při každém spouštění se už spouští strojový kód.
- Skutečný JITer - .NET aplikace je spuštěna a CIL kód je přeložen do strojového kódu. Velkou výhodou je , že tento kód je .NET Frameworkem přímo optimalizovaný na konkrétní stroj, na kterém byl spuštěn. Malou nevýhodou je krátkodobá přítomnost obou verzí programu v paměti stroje, a to strojového a CIL kódu, který je však odstraněn po dokončení kompilace.
- Ekonomický JITer – inteligentní překlad programu tak, že nepřeloží najednou celý program, ale jenom jeho právě prováděnou část. Volání funkcí, které nejsou přeloženy, nahradí tzv. stubem (krátký kód), který navenek vystupuje jako plnohodnotná funkce.

4.1.4 Assemblies

Smyslem assembly (sestavy) je specifikovat logickou jednotku neboli stavební blok pro aplikace .NET, který zapouzdřuje určité vlastnosti. Aplikace pro .NET se skládá z jedné nebo více sestav [2]. Z logického hlediska je sestava sada specifikací. Zejména platí následující:

- Sestava zadává kód (CIL), který je dané sestavě přiřazen. Tento kód se nachází v přenositelném spustitelném souboru.
- Sestava zaručuje seznam datových typů a zajišťuje jmenné prostory pro tyto typy. Každý datový typ v aplikaci .NET musí specifikovat sestavu, do níž patří.
- Sestava určuje pravidla pro určení externích datových typů a externích odkazů včetně odkazů na další sestavy. Tímto způsobem tvoří sestavy hranice oboru názvů odkazu.
- Sestava zadává, které její části jsou vystavené mimo danou sestavu a které jsou pro danou sestavu soukromé.
- Sestava má vlastnosti verze. Všechny prvky sestavy mají své verze jako jednotky – přiřadí se jim čísla verze sestavy, do které náleží.
- Sestava tvoří zaváděcí jednotku. Aplikace pro .NET potřebuje v kterémkoli okamžiku pouze přístup k sestavám, které určují vykonávaný kód. Jiné sestavy tvořící danou aplikaci nemusejí být přítomné, jestliže není jimi specifikovaný kód právě potřebný k vykonávání. Tyto sestavy lze získat na požádání, takže nahrávání aplikací může být výkonnější.

Specifikace v sestavě se souhrnně označují za manifest dané sestavy [2]. Data v manifestu se rovněž označují za metadata. Manifest konkrétně obsahuje:

- název sestavy,
- informace o verzi sestavy,
- informace o zabezpečení sestavy,
- seznam všech souborů, které jsou součástí sestavy,
- informace o typových odkazech pro typy zadané v sestavě,
- seznam dalších sestav odkazovaných danou sestavou.

4.1.5 ADO.NET: Data a XML

ADO.NET (Active Data Object) poskytuje vylepšenou podporu pro oddělené programovací modely a také jim poskytuje podporu pro rozšiřitelný značkovací jazyk (XML). ADO byl vytvořen k zajištění datových služeb tradičním klientským aplikacím, které byly úzce spjaty s databází. ADO.NET byl vytvořen duchu webových aplikací [3].

4.1.6 Webové služby

ASP.NET webové formuláře nabízejí snadný a účinný způsob pro stavbu dynamických uživatelských prostředí. ASP.NET webové služby poskytují stavební bloky pro konstrukci distribuovaných webově založených aplikací. Webové služby jsou založeny na Simple Object

Access Protocol (SOAP) specifikaci. SOAP je protokol pro posílání zpráv XML a je základem webových služeb.

4.1.7 Windows formuláře

Pro aplikace, které jsou založeny na platformě Windows, .NET Framework poskytuje System.Windows.Forms, což je pracovní prostor pro tvorbu uživatelského prostředí.

4.2 .NET Remoting

.NET Remoting podporuje vytváření distribuovaných řešení na platformě .NET. Jde o soubor služeb a technik, které umožňují navázat komunikaci mezi dvěma a více vzdálenými objekty. Vzdálený objekt je chápán jako objekt, který s jinými objekty komunikuje přes hranice své aplikační domény. Distribuované systémy představují systémy typu klient – server. Hlavním úkolem .NET Remotingu je pak zakládat spojení mezi klientem a serverem. Dále čerpáno z literatury [6].

Volat objekt mimo jeho aplikační doménu vyžaduje tento objekt k tomuto účelu uschopnit. Takový proces se nazývá *marschaling* a může probíhat buď hodnotou nebo odkazem. Marschaling hodnotou spočívá v serializaci vzdáleného objektu do proudu bajtů, který se následně deserializuje v aplikační doméně klienta. Klient pak volá tuto kopii vzdáleného objektu lokálně ve své doméně. Marschaling odkazem spočívá ve vytvoření *proxy* objektu v aplikační doméně klienta. Ten pak volání od klienta předává vzdálenému objektu a vrací návratové hodnoty. Klient si tím pádem myslí, že volá server lokálně. Server, který je hostitelem vzdálených objektů, může mít formu konzolové aplikace, služby Windows, webové aplikace i klasické okenní aplikace.

Spojení mezi klientem a serverem je realizováno prostřednictvím speciálních objektů - kanálů. Kanál musí být založen na obou stranách. Platforma .NET poskytuje dva druhy kanálů: TcpChannel a HttpChannel. Kanál HTTP se používá v prostředí sítě internet a kanál TCP v prostředí intranetu. Kanál se skládá z několika částí. Na jeho vstupu je formátovač, který serializuje data do proudu bajtů. Pokud je použit HTTP kanál, tak je nutné požít formátovač typu SOAP. Může být provedeno šifrování zprávy a následně již probíhá fyzický přenos informace mezi aplikačními doménami.

Vzdálené objekty jsou buď *server-activated* (SA) nebo *client-activated* (CA). SA objekty se označují jako *well-known* objekty, které mohou pracovat ve dvou módech: *Singlecall* nebo *Singleton*. Vzdálený SA objekt typu Singleton se vytvoří při první žádosti klienta a existuje jeden pro všechny klienty. Naopak vzdálený objekt typu Singlecall se vytvoří pro každé volání vzdáleného objektu. Před uskutečněním volání vzdáleného objektu je nutné jej zaregistrovat. K tomu je nutné jeho *assembly*, ve které je definována třída objektu, dále *endpoint* a jeho aktivní mód (Singleton nebo Singlecall).

4.3 .NET a projekt Mono

Projekt Mono je sada tzv. open source programů a nástrojů umožňující vývoj a provoz aplikací založených na technologiích .NET i mimo operační systém Windows. Tyto aplikace lze pomocí Mona provozovat i na operačních systémech Unix, Linux, Mac OS X, a Solaris[7]. V Monu jsou implementovány mezinárodní standardy ECMA (European Computer Manufactures Association) použité v platformě .NET a některé základní nástroje:

- běhové prostředí programu (CLI),
- knihovna tříd,
- kompilátor pro jazyk C#.

Mono lze nainstalovat na operační systém Windows společně s .NET Frameworkem, což pro uživatele znamená, že má k dispozici dva podobné prostředky k témuž účelu. Příklad uvádí kompilaci zdrojového kódu v jazyce C# pro obě výše zmíněné varianty:

- csc /out:program_net.exe zdroj.cs (použije MS .NET C# kompilátor),
- mcs –out:program_mono.exe zdroj.cs (použije Mono C# kompilátor).

Složitější je kompilace grafických aplikací vytvořených např. ve Visual Studiu .NET. Ty využívají jmenný prostor `System.Windows.Forms`, který obsahuje sadu prvků specielně navržených pro Windows. Řešení přináší abstraktní třída Gtk, která obsahuje atributy, komponenty, události a metody pojmenovány stejně jako v `System.Windows.Forms`, a tím je zaručena přenositelnost grafických aplikací z Windows na Linux.

Mono se neustále a rychle vyvíjí. Vznikají podpory pro další programovací jazyky a smazává se rozdíl mezi ním a .NET Frameworkem.

5 Porovnání jazyků C++ a C#

Přestože jazyk C++ podporuje orientaci na objekty, nenabízí žádnou formální koncepci rozhraní. Vývojáři v C++ se často musejí uchylovat k abstraktním bázovým třídám a smíšeným rozhraním, když potřebují simulovat programování s rozhraními a spoléhají se na externí modely komponentového programování, jakými jsou COM nebo CORBA [4], jestliže potřebují poskytovat výhody softwaru využívajícího komponenty. Třebaže např. programovací jazyk Java přinesl podporu rozhraní a balíčků na úrovni jazyka, stále ještě obsahoval jen malou podporu vytváření dlouhodobých systémů vycházejících z komponent, v nichž je zapotřebí vyvíjet, propojovat, zavádět a spravovat různé verze komponent z rozličných zdrojů po značnou dobu [2].

Oproti tomu jazyk C# byl již od základů navržen na předpokladu, že moderní systémy se budují pomocí komponent. Proto poskytuje přímou jazykovou podporu obvyklých konstrukcí komponent, jako jsou vlastnosti, metody a události, jež používají nástroje RAD (Rapid Applications Development) k vytváření aplikací z komponent, nastavování vlastností, reagování na události a vzájemnému propojování komponent voláním metod. C# dovoluje vývojářům přímo anotovat a rozšiřovat typové informace komponenty pro zajištění podpory zavedení, návrhu a běhu, integrovat označování verzí komponent přímo do programovacího modelu a integrovat dokumentaci využívající XML přímo do zdrojových souborů. Dále jazyk opouští přístup C++ a COM spočívající v rozdělování zdrojových artefaktů do hlavičkových souborů, implementačních souborů a knihoven typů a dává přednost mnohem jednoduššímu modelu organizace zdroje a opakovaného volání komponent [2].

5.1 Jazyk C#

C# je nový programovací jazyk společnosti Microsoft vytvořený speciálně pro .NET Framework. Jedná se o moderní, objektově orientovaný a typově zabezpečený programovací jazyk odvozený z C a C++, přičemž se C# vyznačuje mnoha syntaktickými podobnostmi s jazyky C++ a Java. Dále čerpáno z literatury [2].

Jako každý moderní objektově orientovaný jazyk podporuje i koncepty, jakými jsou dědičnost, polymorfismus a programování s využitím rozhraní. Podporuje obvyklé konstrukce jazyků C, C++ a Java, jako jsou třídy, struktury, rozhraní a výčty, a také novější konstrukce, jakými jsou delegáti, kteří nabízejí typově zabezpečený ekvivalent ukazatelů na funkce v C/C++, a vlastní atributy, jež umožňují anotování elementů kódů doplňujícími informacemi. Jazyk C# dále zahrnuje prvky z C++, jakými jsou přetěžování operátorů, uživatelem definované konverze a sémantika předávání odkazem.

Na rozdíl od většiny programovacích jazyků nemá jazyk C# žádnou vlastní runtimovou knihovnu. Jazyk C# se se svými potřebami spoléhá na rozsáhlou knihovnu tříd v rámci .NET, včetně práce s konzolí, práce se sítí a soubory. Tato knihovna je implementována právě

v jazyce C#. Ta představuje více než milion řádků kódu a posloužila jako dokonalý zátěžový test během vývojového cyklu jazyka C#, tak i kompilátoru C# [2].

Jazyk C# se snaží o nalezení rovnováhy mezi potřebami konzistentnosti a výkonnosti. Využívá k tomu sjednocený typový systém, v němž jsou všechny typy, včetně primitivních, odvozeny od společného bázového typu, přičemž se zároveň vykonávají optimalizace výkonnosti, jež dovolují zacházet s primitivními typy a jednoduchými, uživatelem definovanými typy, jako s prostou pamětí, což znamená minimální ztrátu a zásadně zvýšenou výkonnost [2].

Jazyk C# je typově zabezpečený jazyk. To znamená, že programy nemohou přistupovat k objektům nevhodnými způsoby. Veškerý kód a data jsou přiřazeny nějakému typu, všechny objekty mají nějaký typ a na objektu lze vykonávat výhradně operace definované určitým přiřazeným typem. Typové zabezpečení odstraňuje celou kategorii chyb v programech z jazyků C a C++ vyplývající z neplatných převodů, nesprávné aritmetiky ukazatelů a dokonce záměrně škodlivého kódu.

Dále jazyk C# poskytuje automatickou správu paměti ve formě velmi výkonného trasovacího generačního mechanismu uvolňování paměti (Garbage Collector). Programátor tak nemusí vykonávat „ruční" správu paměti ani počítání odkazů a eliminuje se celá řada chyb, jakými jsou neplatné ukazatele, paměťové úniky a kruhové odkazy.

5.2 Síťové prostředky v jazyce C#

Knihovna tříd FCL rámce .NET obsahuje řadu typů, které usnadňují přístup k síťovým prostředkům. Tyto typy nabízejí různou úroveň abstrakce a umožňují aplikaci ignorovat mnohé z detailů, které jsou obvykle pro přístup k síťovým prostředkům zapotřebí, přičemž zachovávají vysoký stupeň řízení [2].

5.2.1 System.Net.Sockets

Třídy jmenného prostoru `System.Net.Sockets` implementují standardní API socketů Berkeley pro komunikaci mezi procesy a hostiteli. Poskytují základ pro většinu síťových operací. V tomto oboru názvů jsou připraveny dvě implementace socketů, a to TCP a UDP. Pro práci na úrovni socketů se nejvíce nabízejí třídy `TcpClient` a `TcpListener`, případně `UdpClient`. Popsány budou vybrané prostředky z tohoto oboru názvů.

Třída Socket

Tato třída implementuje standardní socket Berkeley. Každý socket je určen typem socketu (datagram nebo proud) a protokolem, který bude daný socket používat.

Založení socketu:

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

U `ProtocolType` se liší dvě používané hodnoty následovně:

- **Tcp** - vytvoří plně duplexní socket, který, aby jeho prostřednictvím proběhla komunikace, musí být napojen na jiný existující socket metodou `Connect()`,
- **Udp** - vytvoří datagramový socket, který se nespojuje.

U `SocketType` se liší dvě požívané hodnoty následovně:

- **Stream** – pro TCP socket,
- **Dgram –** pro UDP socket.

Nejprve je nutné zvolit IP adresu a port, ke kterým se vytvářený socket bude vztahovat. To se provede následujícím způsobem:

```
IPEndPoint IP_a_Port = new IPEndPoint(IPAddress.Any, 5000);
//IPAddress.Any znamená nespecifikování IP adresy, tzn. kterákoliv IP
```

Dále se socket sváže s tímto koncovým bodem:

```
mujSocket.Bind(IP_a_Port);
```

Navázání spojení v případě protokolu TCP zajišťuje kooperace metod na straně serveru – `Listen()` a `Accept()`:

```
mujSocket.Listen(4); //např. 4 – délka fronty požadavků
mujSocket.Accept(); //přijetí spojení, vrací nový socket
```

a metoda `Connect()` na straně klienta:

```
mujSocket.Connect(IP_a_Port);//zde místo IPAddress.Any konkrétní IP adresa
```

Té musí předcházet zvolení koncového bodu – serveru:

```
IPAddress vzdalenaIPadresa = IPAddress.Parse("147.229.149.112");
IPEndPoint IP_a_Port = new IPEndPoint(vzdalenaIPadresa, 5000);
```

Příklad odeslání dat:

```
byte[] odchoziBajty = new byte[4]; //prostor pro odchozí data - bajty
odchoziBajty = Encoding.ASCII.GetBytes("Ahoj"); //string na byte
mujSocket.Send(odchoziBajty); //odeslání bajtů přes socket
```

Příklad přijetí dat:

```
byte[] prichoziBajty = new byte[1024]; //prostor pro příchozí data - bajty
string prichoziZprava; //prostor pro příchozí data - string
mujSocket.Receive(prichoziBajty); //přijetí proudu bajtů
prichoziZprava = Encoding.ASCII.GetString(prichoziBajty, 0
                 prichoziBajty.Length); //převod byte na string
```

Třída TcpListener

Třída `TcpListener` představuje serverovou abstrakci socketů. Konstruuje se pomocí lokální adresy a portu, k němuž se naváže. Volání metody `Start()` inicializuje naslouchání požadavkům na spojení. Jakmile je obdržen nějaký požadavek, pak buď metoda `AcceptSocket()` nebo `AcceptTcpClient()` přijme připojení a vrátí objekt `Socket` nebo `TcpClient`, který se použije ke komunikaci se vzdáleným klientem. V následující ukázce je založen serverový socket, který příjme data od připojeného klienta:

```
TcpListener server = new TcpListener(5000); //vytvoření server-socketu
server.Start(); //spuštění naslouchání serveru
//po příchodu požadavku na spojení od klienta se vytvoří nový socket:
TcpClient klient = server.AcceptTcpClient();
NetworkStream proud = klient.GetStream(); //vytvoření proudu nad socketem
byte[] prichoziBajty = new byte[1024]; //prostor pro příchozí data - bajty
string prichoziZprava; //prostor pro příchozí data - string
proud.Read(prichoziBajty, 0, prichoziBajty.Length); //přijetí proudu bajtů
prichoziZprava = Encoding.ASCII.GetString(prichoziBajty, 0
                 prichoziBajty.Length); //převod byte na string
proud.Close(); //uzavření proudu
```

Třída TcpClient

Třída `TcpClient` představuje klientskou abstrakci socketů. Ke vzdálenému serveru se připojí pomocí metody `Connect()`. Významnou metodou je `GetStream()`, která umožní odesílání a přijímání dat. V následující ukázce je založen klientský socket, který odešle data na server:

```
TcpClient TcpKlient = new TcpClient("www.nejakaStranka.com", 5000);
//nebo TcpKlient.Connect(IP_a_Port) viz IPEndPoint
//vytvoření streamu nad socketem
NetworkStream proud = TcpKlient.GetStream(); //vytvoření proudu nad sock.
byte[] odchoziBajty = new byte[4]; //prostor pro odchozí data    - bajty
odchoziBajty = Encoding.ASCII.GetBytes("Ahoj"); //převod string na byte
proud.Write(odchoziBajty, 0, odchoziBajty.Length); //odeslání dat
proud.Close(); //uzavření proudu
```

UDP socket

U datagramových socketů se nenavazuje spojení. To znamená, že neexistuje mechanismus dotazu k souhlasu s připojením, jeho odsouhlasení a obsluha, fronta požadavků atd. V tomto režimu můžeme okamžitě odesílat a přijímat pakety od příjemce. Založení UDP socketu:

```
Socket mujSocket = new Socket(AddressFamily.InterNetwork,
                              SocketType.Dgram, ProtocolType.Udp);
```

Příklad odeslání dat:

```
byte[]odchoziBajty = new byte[4]; //prostor pro odchozí data - bajty
odchoziBajty = Encoding.ASCII.GetBytes("Ahoj"); //převod string na byte
IPAddress IPadresaCile = IPAddress.Parse("147.229.149.112");
IPEndPoint IP_a_Port = new IPEndPoint(IPadresaCile, 5000); //IP a port cíle
mujSsocket.SendTo(odchoziBajty, IP_a_Port); //odeslání dat
```

Příklad přijetí dat:

```
byte[]prichoziBajty = new byte[1024]; //prostor pro příchozí data - bajty
IPEndPoint IP_a_Port = new IPEndPoint(IPAddress.Any, 5000);
EndPoint refIP_a_Port = (EndPoint)IP_a_Port; //identifikace odesilatele
mujSocket.Bind(IP_a_Port); //svázaní socketu s lokálním bodem
mujSocket.ReceiveFrom(prichoziBajty, ref refIP_a_Port); příjem dat+IP odes.
string prichoziText = Encoding.ASCII.GetString(prichoziBajty);
```

Při vysílání na broadcastovou IP adresu se koncový bod definuje takto:

```
IPEndPoint broadcastEP = new IPEndPoint(IPAddress.Broadcast, 6000);
```

Při vysílání na multicastovou IP adresu a přijímání dat z této adresy se postupuje takto:
1. vysílač:

```
//adresa multicastové skupiny (z rozsahu D IP adres)
IPAddress IP = IPAddress.Parse("224.5.6.7");
IPEndPoint IPeP = new IPEndPoint(IP, 7000);
byte[]odchoziBajty = new byte[1024]; //prostor pro odchozí data  - bajty
//odeslání zprávy na multicastovou adresu
UDPsocket.SendTo(odchoziBajty, IPeP);
```

2. přijímač:

```
byte[] UDBbbuffer = new byte[1024]; //prostor pro příchozí data - bajty
IPEndPoint IPeP = new IPEndPoint(IPAddress.Any, 7000);
UDPsocket.Bind(IPeP); //svázání socketu s adresou a portem, kde naslouchá
//adresa multicastové skupiny (z rozsahu D IP adres)
IPAddress IP = IPAddress.Parse("224.5.6.7");
//přidání tohoto socketu do multicastové skupiny
UDPsocket.SetSocketOption(SocketOptionLevel.IP,
                          SocketOptionName.AddMembership,
                          new MulticastOption(IP,IPAddress.Any));
UDPsocket.Receive(UDBbbuffer); //příjem multicastové zprávy
```

Pokročilé nastavení socketu

Pokročilé nastavění socketu zajišťuje metoda `SetSocketOption()`, se kterou se pracuje následovně:

```
mujSocket.SetSocketOption(SocketOptionLevel.Socket,
                          SocketOptionName.SendTimeout, 1000)
```

Jejími vstupními parametry jsou dva výčtové typy `SocketOptionLevel` a z hlediska možností zajímavější `SocketOptionName`. Druhý jmenovaný výčet nabízí například možnosti nastavení socketu viz Tab. 5.1.

Tab. 5.1: Výběr pokročilých nastavení socketu

Položka	Význam
SendTimeout	doba čekání na příchod potvrzení
IpTimeToLive	nastavuje velikost hodnoty TTL v hlavičce IP datagramu
DontFragment	povoluje nebo zakazuje fragmentování IP paketu
NoChecksum	nastavuje na UDP datagramu CRC=0
Broadcast	povoluje nebo zakazuje broadcast na socketu
Linger	umožňuje socketu pokračovat v odesílání dat i po zavolání metody `socket.Close()`, a to po nastavenou dobu

5.3 Vlákna v jazyce C#

Vlákna jsou v .NET Frameworku představována třídou `Thread`, která se importuje následovně: `using System.Threading`.

Ve výchozím stavu má obecně každá aplikace jedno vlákno, označované jako primární nebo hlavní vlákno, které vytvoří běhové prostředí CLR. V následujícím příkladu je vytvořeno nové vlákno, ve kterém je spuštěna metoda `NecoUdelej()`, která nemá žádný vstupní parametr, viz ukázka:

```
Thread vlakno = new Thread(new ThreadStart(NecoUdelej)); //vytvoření vlákna
vlakno.Start();//spuštění vlákna

static void NecoUdelej()
{
   //tělo metody, která se spustí v nově vytvořeném vlákně
}
```

V následujícím příkladu je vytvořeno nové vlákno, ve kterém je spuštěna metoda `NecoUdelej()`, která má jeden vstupní parametr, viz ukázka:

```
//vytvoření vlákna
Thread vlakno = new Thread(new ParametrizedThreadStart(NecoUdelej));
vlakno.Start(Paramter); //spuštění vlákna s parametem

static void NecoUdelej(Datovy_typ Nazev_parametru)
{
   //tělo metody, která se spustí v nově vytvořeném vlákně
}
```

K instanci představující aktuální vlákno, v němž se nacházíme, se přistoupí užitím statické vlastnosti `CurrentThread` na tomto objektu. Po použití této vlastnosti je možno zjistit o tomto vlákně různé informace, viz ukázka:

```
public class ThreadInfoExam
{
  internal static void Run()
  {
    Console.WriteLine("ID vlakna : {0}",
    Thread.CurrentThread.GetHashCode());
    Console.WriteLine("Aplikacni domena: {0}",
    Thread.GetDomain().FriendlyName);
    Console.WriteLine("ID aplikacni domeny: {0}",
    Thread.GetDomainID());
```

```
    Console.WriteLine("Priorita vlakna : {0}",
    Thread.CurrentThread.Priority);
    Console.WriteLine("Stav vlakna : {0} ",
    Thread.CurrentThread.ThreadState);
  }
}
```

Každé nově spuštěné vlákno v běhovém prostředí CLR je spuštěno s prioritou *Normal*, což je jedna z hodnot výčtu `ThreadPriority`. Vlastnost *Priority* na instanci vlákna můžeme nejen číst, ale také nastavit na jednu z hodnot zmíněného výčtu. Každé vlákno se nachází v nějakém stavu, který nese vlastnost `ThreadState`. Běžící vlákno má hodnotu této vlastnosti `Running`.

Vlákno je možno na určitou dobu uspat metodou `Sleep()`, viz ukázka:

```
vlakno.Sleep(3000); //vlákno je zablokováno na 3 vteřiny
```

Za účelem synchronizace může mít každý objekt v prostředí .NET svůj monitor, který umožňuje řídit přístup ke specifickým částem zdrojového kódu. Monitor je k dispozici ve jmenném prostoru `System.Threading.Monitor`.

```
Monitor.Enter(this);//začátek zámku
   /*zde se nachází citlivý kód, ke kterému má přístup více vláken, ale
     je zaručeno, že citlivý kód bude vykonán celý bez přerušení právě
     jedním vláknem*/
Monitor.Exit(this);//konec zámku
```

Prakticky tutéž možnost uzamknutí citlivého kódu zaručuje klíčové slovo `lock`:

```
lock(this)
{
   //zde se nachází citlivý kód
}
```

Pro bezpečné provádění např. operací inkrementace, dekrementace je vhodná třída `Interlocked`, která mimo jiné nabízí metodu `Increment()`:

```
Interlocked.Increment(ref pocetKlientu);
```

Klíčové slovo `ref` označuje referenci na proměnnou `pocetKlientu`.

6 Projekt MPKT na platformě .NET

Pro vytvoření programu obsahující rozšířený komunikační protokol MPKT na platformě .NET byl použit programovací jazyk C#, který je moderním objektově orientovaným jazykem vytvořeným primárně pro tuto platformu. Rozšířený komunikační protokol MPKT vychází z protokolu MPKT verze 2008, který byl realizován v rámci praktických cvičení v předmětu Pokročilé komunikační techniky.

6.1 Popis nového projektu

Projekt se skládá ze dvou samostatných aplikací, a to z klientské a serverové aplikace a dále ze společné knihovny tříd, která obsahuje třídy a metody užívané oběma aplikacemi. Obě aplikace mají grafické uživatelské rozhraní s intuitivním ovládáním. Projekt respektive protokol obsahuje vylepšení uvedena v kapitole 3.1. Společná knihovna tříd je představována dynamicky linkovanou knihovnou s příponou .dll, která je do obou aplikací přidána následující konstrukcí: `using KnihovnaTrid`.

Tato knihovna obsahuje definici tříd Paket, Protokol, CRC16 a jejich metod. Tyto třídy využívají programové prostředky z následujících jmenných prostorů:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Net;
using System.Net.Sockets;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization;
using System.IO;
```

6.1.1 Třída Paket

Ve třídě Paket jsou definovány položky, které představují námi definovanou hlavičku a tělo přenášené textové zprávy, viz ukázka:

```
[Serializable] // třídu je možno serializovat do proudu bajtů
public class Paket
{
  public byte     CisloKlienta;
  public ushort   Offset;
  public ushort   DelkaZpravy;
  public bool     MoreFragments;
  public eOporace Operace;
  public ushort   CRC16;
  public string   Zprava;
  //konstruktor
  public Paket(byte ck, ushort of, ushort dz, bool mf, eOporace op,
               ushort crc, string zp)
  {
    this.CisloKlienta = ck;
    this.Offset = of;
    this.DelkaZpravy = dz;
```

```
        this.MoreFragments = mf;
        this.Operace = op;
        this.CRC16 = crc;
        this.Zprava = zp;
    }
```

Jak je patrné z defice třídy Paket, tak námi definovaný komunikační protokol bude pracovat s následujícími položkami hlavičky dle následující tabulky:

Tab. 6.1: Položky hlavičky nového paketu MPKT

Atribut	Význam
CisloKlienta	jedinečné číslo klienta z rozsahu 0-255
Offset	je-li zpráva rozdělena do více částí, představuje posun od počátku zprávy
DelkaZpravy	obsahuje délku zprávy, případně délku fragmentu, je-li zpráva delší
MoreFragments	nastaveno na true, je-li zpráva rozdělena do více fragmentů, jinak false
Operace	obsahuje typ přenášené zprávy
CRC16	obsahuje kontrolní součet zprávy, případně fragmentu

č. klienta 1B	offset 2B	délka zpr. 2B	MoreFragments 1B	operace 4B	CRC 2B	0-100B
hlavička						zpráva

Obr. 6.1: Nový paket MPKT

Možné typy přenášených zpráv ve výše uvedené třídě Paket jsou sdruženy do výčtového datového typu `enum`, který obsahuje tyto položky:

```
public enum eOporace
{
  TextMessage,  // textová zpráva
  bCastMessage, // textová zpáva pro broadcastové přeposlání
  Init,         // inicializační zpráva
  AccessDenied, // žádost o spojení zamítnuta, duplicita ID klienta
  Acknowledge,  // kladné potvrzení
  End,          // žádost o ukončení spojení
  PacketDenied   // příchozí paket odmítnut z důvodu chyby ve zprávě
}
```

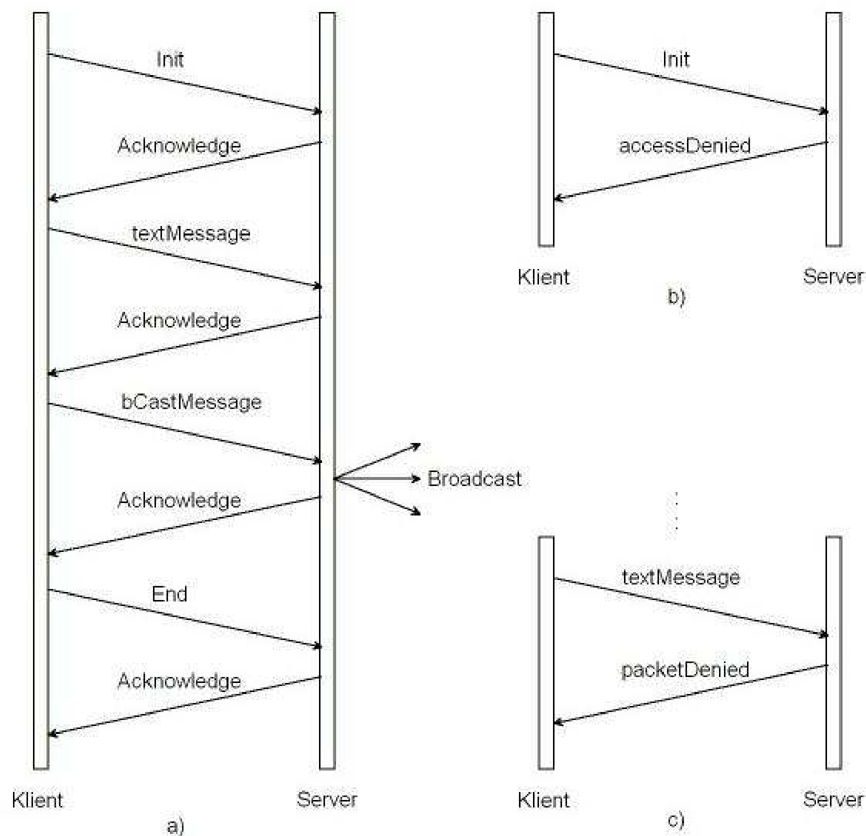
Schéma komunikace mezi klientskou a serverovou aplikací bude podle výše uvedených typů zpráv vypadat podle obrázku 6.2.

6.1.2 Třída Protokol

Ve třídě Protokol jsou definovány metody pro obsluhu komunikace mezi klientskou a serverovou aplikací, viz tabulka 6.2.

Tab. 6.2: Metody třídy Protokol

Metoda	Význam
NaslouchejTCP()	server naslouchá příchozím požadavkům na TCP spojení
NavazSpojeniTCP()	klient vysílá žádost na server o navázání TCP spojení
PosliTCPdata()	zajišťuje sestavení paketu a jeho odeslání směrem k cíli
PrijmiTCPdata()	zajišťuje přijetí paketu a kontrolu textové zprávy podle hlavičky paketu
KontrolaZpravy()	podílí se na výše zmíněné kontrole textové zprávy



Obr. 6.2: Schéma komunikace klient – server
a) normální stav, b) žádost o spojení zamítnuta, c) detekována chyba ve zprávě

Následuje podrobný popis metod třídy Protokol:

```
public Socket NaslouchejTCP(TcpListener server)
{
  Socket Klient;
  try
  {
      Klient = server.AcceptSocket(); //socket pro kumunikaci s klientem
      return Klient;
  }
  catch (Exception){ return null; } //zachycení chyby/výjimky
}
```

Metoda `NaslouchejTCP()` slouží serverové aplikaci pro naslouchání příchozím požadavkům na TCP spojení. Jejím vstupním parametrem je objekt typu `TcpListener`, který představuje serverový socket. Jeho metoda `AcceptSocket()` je blokovací, tzn. že čeká, dokud nepřijde požadavek na TCP spojení a přijde-li takový požadavek, vrací nový socket, na kterém bude probíhat komunikace s právě nově připojeným klientem. V případě nastanuvší chyby při provádění této operace je tato chyba zachycena pomocí výrazu `catch(Exeption)`.

```
public TcpClient NavazSpojeniTCP(string IP, string port)
{
  TcpClient klient;
  IPAddress serverIP = IPAddress.Parse(IP); //IP adresa serveru
  int serverPort = Convert.ToInt16(port); //číslo TCP portu serveru
  IPEndPoint endPoint = new IPEndPoint(serverIP, serverPort);
```

```
  try
  {
    klient = new TcpClient(); //nová instance klientského socketu
    klient.Connect(endPoint); //pokus o připojení na server
    return klient;
  }
  catch (Exception){ return null; }//zachycení chyby/výjimky
}
```

Metoda `NavazSpojeniTCP()` slouží klientské aplikaci pro navázání TCP spojení se serverem. Vstupními parametry této metody jsou textové řetězce IP a port, které představují IP adresu cílového serveru a číslo portu, na kterém server naslouchá příchozím požadavkům na navázání TCP spojení. Nejprve je provedena konverze zmíněných řetězců na patřičné datové typy a je vytvořen objekt typu `IPEndoPoint`, který obsahuje IP adresu a port cílového serveru. Je vytvořen objekt typu `TcpClient`, který představuje klientský socket. Jeho metoda `Connect()` se pokusí o navázání TCP spojení se vzdáleným serverem. V případě úspěchu je vrácen socket s navázaným spojením, jinak je pomocí `catch(Exeption)` zachycena chyba.

Následuje popis metody, která slouží k odesílání zpráv:

```
public void PosliTCPdata(byte id, Socket Socket, eOporace operace, string
                          zprava)
{
  byte[] poleBajtu = new byte[512]; //prostor pro odchozí data - bajty
  MemoryStream mujStream = new MemoryStream(); //stream pro serializaci
  ushort delka = (ushort)zprava.Length; //délka odchozí zprávy
  ushort crc;
  if (delka != 0){
      Crc16 mojeCRC16 = new Crc16(); //instance pro výpočet CRC
      crc = mojeCRC16.Checksum(zprava); } //vypočítá se CRC odchozí zprávy
  else { crc = 0; }
  //naplnění položek paketu
  Paket mujPaket = new Paket(id, delka, operace, crc, zprava);
  IFormatter Formator = new BinaryFormatter(); //potřebný pro serializaci
  //serializace paketu na objekt typu stream
  Formator.Serialize(mujStream, mujPaket);
  poleBajtu = mujStream.ToArray(); //převod streamu na byte
  Socket.Send(poleBajtu); //odeslání paketu jako proud bajtů
}
```

Metoda `PosliTCPdata()` slouží k odesílání zpráv mezi klientskou a serverovou aplikací v obou směrech. Pro větší názornost je zde uvedena odlehčená metoda, která nebere v úvahu fragmentaci textové zprávy a postupné odeslání jednotlivých fragmentů. Včetně fragmentace je tato metoda uvedena v **příloze B**. Vstupními parametry této metody jsou identifikační číslo klienta, socket, na kterém probíhá TCP komunikace, typ zprávy neboli operace a textová zpráva, která však nemusí být v paketu vůbec obsažena, bude-li se jednat a potvrzovací nebo zamítací paket. První úlohou této metody je vytvoření instance třídy Paket a naplnění jejích atributů. Následně je tato instance podrobena technice serializace, která obecně převádí instance tříd do proudu bajtů, ke kterým jsou navíc přidány potřebné režijní informace, aby bylo možné z proudu bajtů opět sestavit požadovanou instanci třídy. Vlastní serializaci musí předcházet vytvoření objektů typu `MemoryStream` a typu `BinaryFormatter`, jehož metoda

Serialize(mujStream, mujPaket) provede samotnou serializaci. Pak se provede převod serializovaného objektu na datový typ byte způsobem poleBajtu=mujStream.ToArray() a ten se předá socketu k odeslání. Výsledný datový proud po serializaci obsahuje nejen data náležející našemu paketu, ale také potřebné režijní informace nutné pro správnou deserializaci. Délka těchto režijních dat je kolem 260B, což je pro nás sice přenosová zátěž navíc, ovšem je zaručena veškerá paměťová bezpečnost na rozdíl od nezabezpečeného způsobu kopírování bloků paměti s využitím ukazatelů. Tento způsob převodů struktur na pole bajtů byl v tomto případě také testován, ovšem nepodařilo se zajistit, aby se aplikace chovaly zcela korektně.

Následuje popis metody, která slouží k přijetí zprávy:

```
public Paket PrijmiTCPdata(Socket Socket)
{
  byte[] poleBajtu = new byte[512]; //místo pro příchozí data - bajty
  MemoryStream mujStream = new MemoryStream(); //stream pro deserializaci
  Paket mujPaket = null; //instance paketu pro naplnění příchozími daty

  Socket.Receive(poleBajtu); //příjem zprávy - proud bajtů
  //Zápis příchozích bajtů do objektu typu stream
  mujStream.Write(poleBajtu, 0, poleBajtu.Length);
  IFormatter lFormatter = new BinaryFormatter(); //nutný pro deserializaci
  mujStream.Position = 0; //pozice ve streamu na počátek
  //deserializace proudu bajtů na objekt třídy Paket
  mujPaket = (Paket)lFormatter.Deserialize(mujStream);
  mujStream.Flush(); //uvolní mujStream

  if ((mujPaket.Operace == eOporace.TextMessage) || (mujPaket.Operace ==
       eOporace.bCastMessage))
  {
    if (KontrolaZpravy(mujPaket)) return mujPaket; //kontrola bezchybnosti
    else return null;
  }
  else return mujPaket; //jedná se o paket neobsahující text. zprávu
}
```

Metoda PrijmiTCPData() slouží k příjmu zpráv na klientské a serverové aplikaci. Pro větší názornost je zde uvedena odlehčená metoda, která nebere v úvahu fragmentaci textové zprávy a její zpětné složení na straně příjemce. Včetně fragmentace je metoda uvedena v **příloze C**. Vstupním parametrem této metody je socket, na kterém probíhá TCP komunikace. Výstupem je instance třídy Paket. Socket metodou Receive() přijme prostý proud bajtů. Poté je nutné provést deserializaci, která z proudu bajtů zpětně vytváří instanci třídy Paket. Vlastní deserializaci musí předcházet vytvoření objektů typu MemoryStream, do kterého se zapíše příchozí proud bajtů a pozice v tomto streamu se nastaví na počátek, a typu BinaryFormatter, jehož metoda Deserialize(mujStream) provede samotnou deserializaci. Přenáší-li paket textovou zprávu, tak se provede kontrola této zprávy na případnou chybu nastanuvší během přenosu zprávy. K této kontrole slouží privátní metoda KontrolaZpravy(), jejímž vstupním parametrem je objekt typu Paket. Kontrola paketu se provádí na základě délky textové zprávy a jejího kontrolního redundantního součtu. Metodu Checksum() pro

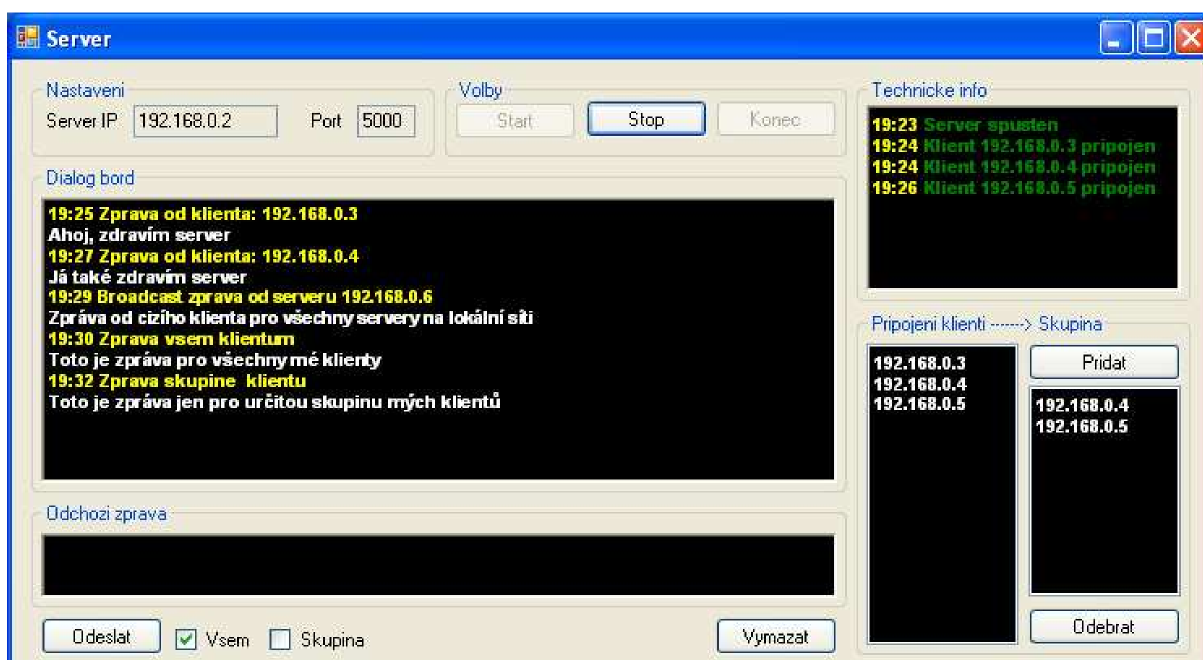
výpočet CRC poskytuje třída Crc16. Odpovídájí-li údaje délka zprávy a CRC uvedené v hlavičce paketu s vypočtenými údaji na straně příjemce, je zpráva považována za bezchybnou a je navrácena hodnota true.

```
private bool KontrolaZpravy(Paket paket)
{
  Crc16 mojeCRC16 = new Crc16(); //instance pro výpočet CRC
  ushort crc = mojeCRC16.Checksum(paket.Zprava); //CRC příchozí zprávy
  ushort delka = (ushort)paket.Zprava.Length; //délka příchozí zprávy

  if ((delka == paket.DelkaZpravy) && (crc == paket.CRC16)) return true;
  else return false;
}
```

6.2 Server – popis funkce

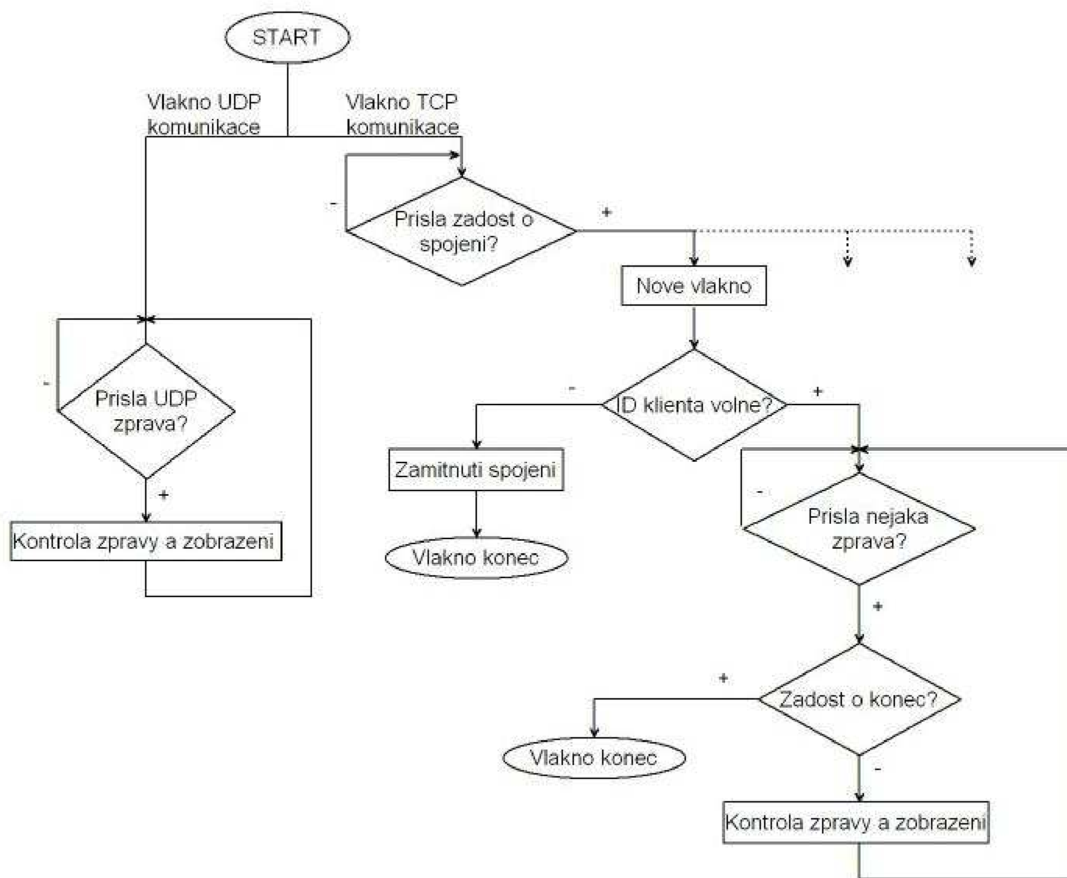
Server je vytvořen jako formulářová aplikace, která nabízí pohodlné ovládání svého chodu. Server je schopen komunikovat jak prostřednictvím TCP socketu tak i UDP socketu. Plný zdrojový kód včetně komentářů je k dispozici na přiloženém CD. Zde je uveden popis metod týkajících se problematiky socketů, vedení komunikace a obsluhy vláken, takže příkazy, které nejsou významně důležité, jsou z popisu vypuštěny.

Tlačítkem *Start* se vytvoří dvě vlákna programu, přičemž první vlákno slouží k obsluze požadavků na TCP spojení od klientů a druhé vlákno slouží k obsluze příchozích broadcastových zpráv přes UDP socket. Na obrázku 6.3 je zachycena serverová aplikace.



Obr.6.3: Serverová aplikace

Chování serveru při komunikaci s jedním klientem zachycuje vývojový diagram, který je uveden na obrázku 6.4. Na vlákně pro UDP komunikaci server naslouchá možným příchozím zprávám od ostatních serverů.



Obr.6.4: Chování serveru při komunikaci s klientem

O naslouchání příchozích požadavků na TCP spojení se stará metoda `StartTCPlistening()`, která je spuštěna v prvním vlákně, viz ukázka:

```
public void StartTCPlistening()
{
  mujProtokol = new Protokol(); //instance vlastní třídy Protokol
  TcpListener server; //serverový socket
  Socket mujTCPklient; //socket pro klienta
  ArrayList seznamSocketu = ArrayList.Synchronized(new ArrayList());
  Thread VlaknoTCPklient; //vlákno pro nového klienta
  int port = Convert.ToInt16(textBox_port.Text); //načtení čísla portu
  server = new TcpListener(IPAddress.Any, port); //vytvoření server-socketu
  server.Start(); //server naslouchá
  UpdateTechInfo(" Server spusten\n", 'Z');
  while (true) //smyčka pro naslouchání požadavků na spojení
  {
    mujTCPklient = mujProtokol.NaslouchejTCP(server); //čekání na požadavek
    if (mujTCPklient != null)
    {
      seznamSokcetu.Add(mujTCPklient); //přidá socket do seznamu socketů
      VlaknoTCPklient =
      new Thread(new ParameterizedThreadStart(ObsluzTCPklienta));
      VlaknoTCPklient.Start(mujTCPklient); //spuštění vlákna pro klienta
    }
    else break; //ukončení smyčky v případě ukončení naslouchání serveru
  }
}
```

V metodě `StartTCPlistening()` se nejprve vytvoří instance třídy Protokol, která zpřístupní metody pro obsluhu komunikace mezi serverem a jeho klienty. Dále je vytvořen objekt typu `TcpListener`, který představuje serverovou abstrakci socketu. Je svázán s lokální IP adresou a zadaným číslem portu. Naslouchání příchozích požadavků na TCP spojení spouští metoda `Start()`. Server v nekonečné smyčce v blokovacím režimu socketu čeká na příchozí požadavek na spojení a jakmile takový požadavek od nějakého klienta přijde, je vytvořen nový socket pro komunikaci s tímto klientem. Pro každého nově připojeného klienta se vytvoří nové vlákno, ve kterém se spouští metoda `ObsluzTCPklienta()`, jejímž vstupním parametrem je onen nově vytvořený socket pro komunikaci s tímto klientem, viz dále:

```
public void ObsluzTCPklienta(object klient)
{
  Socket klientVeVlaknu = (Socket)klient; //socket komunikace s klientem
  Paket mujPaket = null; //instance paketu pro naplnění příchozími daty
  byte idTohotoKlienta; //identifikační číslo klienta
  //čekání na inicializační zprávu s ID klienta
  mujPaket = mujProtokol.PrijmiTCPdata(klientVeVlaknu);
  if (mujPaket != null)
  {
    if (!PridejKlienta(mujPaket.CisloKlienta)) //kontrola duplicity ID
    {
      operace = eOporace.AccessDenied;
      mujProtokol.PosliTCPdata(mujPaket.CisloKlienta, klientVeVlaknu,
                               operace, ""); //zamítací zpráva
      UzavriSocket(klientVeVlaknu, 256); //uzavření spojení s klientem
      return;
    }
    else //ID klienta je schváleno, odešle se kladné potvrzení
    {
      operace = eOporace.Acknowledge;
      mujProtokol.PosliTCPdata(mujPaket.CisloKlienta, klientVeVlaknu,
                               operace, "");
    }
  }
  idTohotoKlienta = mujPaket.CisloKlienta;
  while (true) //je vedena komunikace s klientem, blokovací režim socketu
  {
    mujPaket = mujProtokol.PrijmiTCPdata(klientVeVlaknu); //příjem zprávy
    if ((mujPaket != null) && (mujPaket.CisloKlienta == idTohotoKlienta))
    { //v paketu nebyla detekována chyba CRC ani délky zprávy
      switch (mujPaket.Operace) //jakého je příchozí zpráva typu?
      {
        case eOporace.TextMessage:
        ZobrazZpravu(mujPaket, IPadresaStr);
        operace = eOporace.Acknowledge;
        break;
        case eOporace.bCastMessage:
        ZobrazZpravu(mujPaket, IPadresaStr);
        PosliUDPvsemServerum(UDPsocketVys, mujPaket.Zprava);
        operace = eOporace.Acknowledge;
        break;
        case eOporace.End: operace = eOporace.Acknowledge; break;
        default: operace = eOporace.PaketDenied; break;
      }
    }
    else operace = eOporace.PaketDenied; //v paketu detekována chyba
    mujProtokol.PosliTCPdata(idTohotoKlienta, klientVeVlaknu, operace, "");
    if (mujPaket.Operace == eOporace.End) break; //ukončení smyčky
```

```
  }
  UzavriSocket(klientVeVlaknu, idTohotoKlienta); } //ukončení komunikace
```

V metodě ObsluzTCPklienta() se nejprve porovná identifikační číslo klienta s již obsazenými čísly ostatních připojených klientů, viz metoda PridejKlienta():

```
private bool PridejKlienta(byte id)
{
  if (!seznamKlientu.Contains(id)) //zda toto ID v seznamm ještě není
  {
    seznamKlientu.AddLast(id); //toto ID se vloží do dynamického seznamu
    Interlocked.Increment(ref pocetKlientu); chráněná inkrementace
    return true;
  }
  else return false; //pokud je zjištěna duplicita ID
}
```

V případě duplicity ID klienta je klient informován zprávou typu eOporace.AccessDenied, že jeho žádost o spojení je zamítnuta a tato relace je uzavřena. V opačném případě je klient informován zprávou typu eOporace.Acknowledge a je zařazen do dynamického seznamu identifikačních čísel klientů. Poté server v nekonečné smyčce v blokovacím režimu socketu čeká na příchozí textové zprávy od tohoto klienta. Zpráva je přijata a zkontrolována: mujPaket=mujProtokol.PrijmiTCPdata(klientVeVlaknu). Textová zpráva je zobrazena metodou ZobrazZpravu() na příslušné místo ve formuláři, případně je odeslána na broadcastovou adresu dané lokální sítě ostatním serverům, je-li příchozí zpráva typu eOporace.bCastMessage a klientovi je odesláno kladné potvrzení. Přijde-li zpráva typu eOporace.End, klient žádá o ukončení TCP spojení, které obslouží metoda UzavriSocket(), jejímž vstupními parametry jsou socket a ID daného klienta, viz ukázka:

```
public void UzavriSocket(Socket pomocnySocket, byte id)
{
  pomocnySocket.Close(); //uzavření socketu
  //odstranění socketu z seznamu socketů
  seznamSocketu.Remove(pomocnySocket);
  seznamKlientu.Remove(id); //uvolní ID klienta ze seznamu ID
  Interlocked.Decrement(ref pocetKlietnu); //dekrementuje se počet klientů
  UpdateClientList(); //update seznamu připojených klientů pro info
}
```

Je-li třeba server pozastavit, je nutné uzavřít navázaná TCP spojení se všemi klienty. K pozastavení serveru slouží tlačítko *Stop* a potřebné úkony s ukončováním navázaných spojení zajistí metoda UzavriVsechnySockety():

```
public void UzavriVsechnySockety()
{
  Socket pomocnySocket;
  for (int i = 0; i < seznamSocketu.Count; i++) //procházení seznamem
  {
    pomocnySocket = (Socket)seznamSocketu[i];
    if (pomocnySocket != null)
    {
      if (pomocnySocket.Connected) //zda je spojení stále aktivní
```

```
        {
          pomocnySocket.Close(); //socket se uzavře a uvolní se prostředky
          pomocnySocket = null;
        }
      }
  } //následuje uzavření všech serverových socketů, vyprázdnění seznamů
  UDPmultiCastSocket.Close();
  UDPsocketVys.Close();
  UDPsocketPrijem.Close();
  seznamSocketu.Clear();
  seznamKlientu.Clear();
  pocetKlientu = 0;
}
```

Všechny sockety jsou uloženy v objektu typu `ArrayList`, jehož položky jsou dostupné přes index. Pomocí metody socketu `Close()` se dané spojení uzavře a uvolní se veškeré prostředky spojené s touto relací. Dále se uzavřou všechny 3 UDP sockety, které budou popsány dále. Nakonec je také vyprázdněn seznam identifikačních čísel klientů. O takto uzavřeném spojení se klientská aplikace dozví testováním dostupnosti socketu před každým vysíláním, že spojení bylo ukončeno vzdáleným strojem.

Přijde-li od klienta zpráva typu `eOporace.bCastMessage`, je tato zpráva standardně zkontrolována jako obyčejná textová zpráva, avšak poté je předána metodě `PosliUDPvsemServerum()`, jejímiž vstupními parametry jsou UDP socket a textová zpráva. Tato metoda zajistí, že textová zpráva bude odeslána na broadcastovou adresu a přijmou ji všechny ostatní servery v této lokální síti přes UDP socket na portu 6000. Následuje popis této metody:

```
public void PosliUDPvsemServerum(Socket pomocnySocket, string text)
{
  byte[] odchoziBajty = new byte[1006]; //prostor pro odchozi data - bajty
  IPEndPoint broadcastEP = new IPEndPoint(IPAddress.Broadcast, 6000);
  text = text.Insert(0, Prefix); //přidání prefixu ke zprávě
  odchoziBajty = Encoding.ASCII.GetBytes(text); //převod string na byte
  pomocnySocket.SendTo(odchoziBajty, broadcastEP); //odeslání zprávy
}
```

Tlačítkem *Start* se kromě již popsaného naslouchání příchozích požadavků na TCP spojení spouští druhé vlákno, které obsluhuje příchozí zprávy přes UDP socket na portu 6000. O tuto obsluhu se stará metoda `StartUDPlistening()`, viz ukázka:

```
public void StartUDPlistening()
{ //vytvoření socketů pro UDP komunikaci
  Socket UDPsocketPrijem = new Socket(AddressFamily.InterNetwork,
                              SocketType.Dgram, ProtocolType.Udp);
  Socket UDPsocketVys = new Socket(AddressFamily.InterNetwork,
                            SocketType.Dgram, ProtocolType.Udp);
  Socket UDPmultiCastSocket = new Socket(AddressFamily.InterNetwork,
                                  SocketType.Dgram, ProtocolType.Udp);
  byte[] UDPbbuffer = new byte[1006]; //prostor pro příchozí data - bajty
  string[] jenIP;
  IPEndPoint UDPhost = new IPEndPoint(IPAddress.Any, 6000);
  EndPoint refUDPhost = (EndPoint)UDPhost; //sem se uloží IP a port odes.
```

```
  UDPsocketPrijem.Bind(UDPhost); //svázání socketu s lokální IP a portem
  while (true) //smyčka pro naslouchání
  {
    UDPsocketPrijem.ReceiveFrom(UDPbbuffer, ref refUDPhost);//příjem zprávy
    string ip = refUDPhost.ToString(); //převod IP adresy na string
    string prichoziText = Encoding.ASCII.GetString(UDBbbuffer);
    if (prichoziText.StartsWith(Prefix)) //měl by být 0xF0F0
    {
      prichoziText = prichoziText.Remove(0, 6); //odstranění prefixu
      jenIP = ip.Split(':');
   DialogBordBox(" Broadcast zprava od serveru: ", prichoziText, jenIP[0]);
    }
  }
}
```

V metodě `StartUDPlistening()` se nejprve vytvoří tři UDP sockety. První slouží pro příjem broadcastových zpráv od ostatních serverů, druhý slouží pro vysílání broadcastových zpráv ostatním serverům a třetí socket slouží k vysílání multicastových zpráv klientům. První socket je svázán s lokální IP adresou a portem metodou `UDPsocketPrijem.Bind(UDPhost)` a dále v nekonečné smyčce čeká v blokovacím režimu socketu na příchozí zprávu, a to zajišťuje metoda socketu `UDPsocketPrijem.ReceiveFrom(UDBbbuffer, ref refUDPhost)`, jejímž prvním parametrem je místo pro uložení příchozího proudu bajtů a druhý parametr slouží pro uložení IP adresy odesilatele zprávy. Zkontroluje se, zda příchozí zpráva je skutečně určena naší aplikaci, a to kontrolou prefixu příchozí zprávy. Poté je zpráva zobrazena.

Posílání textových zpráv ze serveru směrem k jeho klientům je provedeno přes UDP socket. Odeslání textové zprávy všem připojeným klientům umožňuje skutečnost, že každý klient je po akceptování žádosti o spojení od serveru umístěn do multicastové skupiny, jejíž IP adresu server zná. Klient naslouchá v samostatném vlákně na UDP socketu takovýmto příchozím multicastovým zprávám (více viz popis klientské aplikace). O odeslání multicastové zprávy ke klientům se na serveru stará metoda `PosliMultiZpravu()`, viz ukázka:

```
public void PosliMultiZpravu()
{
  byte[] odchoziBajty = new byte[1006];//prostor pro odchozí data - bajty
  string odchoziText;
  IPAddress IP = IPAddress.Parse("224.5.6.7");//adresa multicastové skupiny
  IPEndPoint IPeP = new IPEndPoint(IP, 7000);//koncový bod..IP adresa, port

  odchoziText = richTextBox2.Text + "\n";//načtení uživ. zadané zprávy
  odchoziText = odchoziText.Insert(0, Prefix);//přidání prefixu ke zprávě
  odchoziBajty = Encoding.ASCII.GetBytes(odchoziText);//převod string->byte
  //odeslání multicastové zprávy
  UDPmultiCastSocket.SendTo(odchoziBajty, IPeP);//odeslání zprávy klientům
}
```

Nejprve je vytvořen koncový bod, představovaný multicastovou IP adresou a číslem portu. Dále je k odchozí textové zprávě přidán prefix, který zajišťuje příslušnost k naší aplikaci. Zpráva je odeslána metodou socketu `SendTo()`.

Je také dále možné zaslat textovou zprávu jen jednomu klientovi nebo více vybraným klientům. Označením IP adresy požadovaného klienta a stiskem tlačítkem *Pridat* je IP adresa tohoto klienta přidána do seznamu *Skupina klientů*. Zaškrtnutím volby *Skupina* a stiskem tlačítka *Odeslat* se uskuteční odeslání textové zprávy přes UDP socket jen zvolené skupině klientů, což zajišťuje metoda `PosliUDPKlientum()`, viz ukázka:

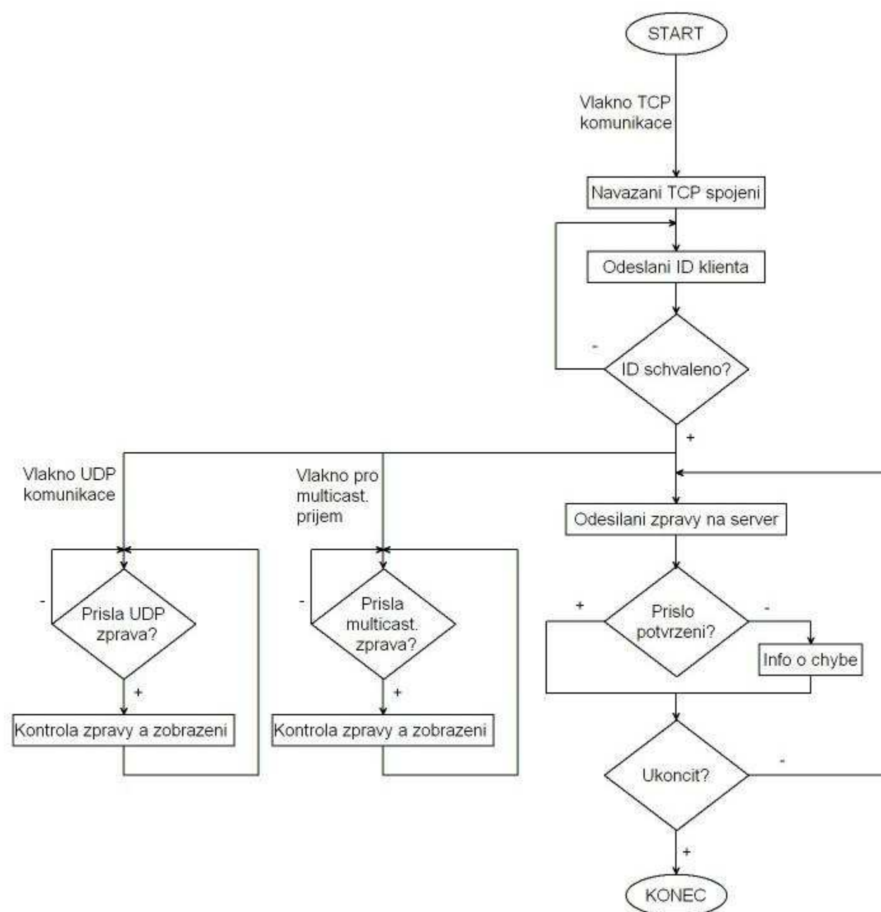
```
public void PosliUDPKlientum()
{
  byte[] odchoziBajty = new byte[1006];//prostor pro odchozí data - bajty
  string odchoziText;
  Socket SkupinaSocket = new Socket(AddressFamily.InterNetwork,
                                    SocketType.Dgram, ProtocolType.Udp);
  for (int i = 0; i < listBoxSkupina.Items.Count; i++)
  {
    IPAddress IP = IPAddress.Parse(listBoxSkupina.Items[i].ToString());
    IPEndPoint IPeP = new IPEndPoint(IP, 8000); );//IP adresa, port klienta

    odchoziText = richTextBox2.Text + "\n";//načtení uživ. zadané zprávy
    odchoziText = odchoziText.Insert(0, Prefix);//přidání prefixu ke zprávě
    odchoziBajty = Encoding.ASCII.GetBytes(odchoziText);//string na byte
    SkupinaSocket.SendTo(odchoziBajty, IPeP); //odeslání zprávy klientům
  }
  SkupinaSocket.Close();//uzavření socketu
}
```

V metodě `PosliUDPKlientum()` je vytvořen UDP socket, přes který budou odesílány textové zprávy k vybraným klientům. Klient naslouchá v samostatném vlákně na UDP socketu takovýmto příchozím zprávám (více viz popis klientské aplikace). Poté je v cyklu prováděn nejprve převod IP adresy požadovaného klienta ze seznamu vybraných klientů z datového typu string na objekt `IPEndPoint`. Dále je k odchozí textové zprávě přidán prefix, který zajišťuje příslušnost k naší aplikaci. Zpráva je odeslána metodou socketu `SendTo()`. Tento cyklus se opakuje pro všechny vybrané klienty.

6.3 Klient – popis funkce

Klient je vytvořen jako formulářová aplikace, která nabízí pohodlné ovládání svého chodu. Je primárně určen pro komunikaci s jedním serverem ve stejné lokální síti. Spojení však lze navázat i v síti internet. Klientská aplikace je rovněž vícevláknová. V hlavním vlákně programu je vedena TCP komunikace se serverem. Dále jsou vytvořena dvě vlákna, která slouží pro příjem unicastových a multicastových UDP zpráv zasílaných serverem. Plný zdrojový kód klientské aplikace včetně komentářů je k dispozici na přiloženém CD.

Chování klientské aplikace při komunikaci se serverem zachycuje vývojový diagram, který je uveden na obrázku 6.5.



Obr. 6.5: Chování klientské aplikace při komunikaci se serverem

Následuje popis jednotlivých metod klientské aplikace, který je zaměřen především na problematiku socketů, vedení komunikace a obsluhu vláken, takže příkazy, které nejsou významně důležité, jsou z popisu vypuštěny.

Nejprve je potřeba ve formuláři správně vyplnit IP adresu serveru a příslušné číslo portu, kde server naslouchá příchozím požadavkům na TCP spojení. Dále si každý klient musí zvolit své identifikační číslo z rozsahu 0-255. Poté se již může tlačítkem *Pripojit* pokusit o navázání TCP spojení se serverem. Pokud zvolené identifikační číslo již není obsazené jiným klientem serveru, tak klient může očekávat kladné potvrzení o úspěšném navázání spojení se serverem. V opačném případě musí klient zvolit jiné ID a proces opakovat. Metoda `button_start_Click()` je vyvolána stiskem tlačítka *Pripojit*, viz ukázka:

```
private void button_start_Click(object sender, EventArgs e)
{
  cisloKlienta = Convert.ToByte(textBox_ID.Text);//ID klienta
  try
  {
    mujProtokol = new Protokol();//nová instance třídy Protokol
    Socket TCPsocket = new Socket(AddressFamily.InterNetwork,
                                  SocketType.Stream, ProtocolType.Tcp);
    TCPsocket.ReceiveTimeout = 10000;//limit čekání socketu na potvrzení
    int port = System.Convert.ToInt16(textBox_port.Text);
    IPAddress IP = IPAddress.Parse(textBox_IP.Text);//získání IP ze stringu
    IPEndPoint IPeP = new IPEndPoint(IP, port);//IP adresa a port serveru
```

```
    TCPsocket.Connect(IPeP);//pokus o spojení se serverem
    operace = eOporace.Init;//inicializační typ zprávy
    //odeslání inicializační zprávy, poté čekání na potvrzení
    mujProtokol.PosliTCPdata(cisloKlienta, TCPsocket, operace, "");
    paketOdServeru = mujProtokol.PrijmiTCPdata(TCPsocket);
    if ((paketOdServeru != null) && (paketOdServeru.Operace ==
                                     eOporace.Ackonwledge))
    {
      label_info.Text = "Server akceptuje zadost spojeni";
      PrijemUDPthread = new Thread(new ThreadStart(cekejMultiZpravu));
      PrijemUDPthread.Start();//nové vlákno, příjem multicast. UDP zpráv

      PrijemTCPthread = new Thread(new ThreadStart(CekejUDPZpravu));
      PrijemTCPthread.Start();//nové vlákno, příjem unicast. UDP zpráv
    }
    else { label_info.Text = "Server zamitnul zadost o spojeni"; }
  }
  catch (Exception)//zachycení výjimky socketu
  {
    label_info.Text = "Vyprsel casovy limit zadosti o spojeni";
  }
}
```

V metodě `button_start_Click()` se vytvoří instance třídy protokol, která zpřístupní metody pro obsluhu komunikace mezi klientem a serverem. Dále je vytvořen TCP socket a jeho vlastnost `ReceiveTimeout` nastavena na 10s, což znamená, že pokud do deseti sekund nepřijde žádná zpráva, v tomto případě potvrzení o akceptování spojení od serveru, tak socket zachytí chybu. Jde tedy o časovač. Dále jsou IP adresa a číslo TCP portu serveru převedeny na příslušné objekty a metodou `TCPsocket.Connect(IPeP)` je navázáno spojení se serverem. Poté je na server odeslána zpráva typu `eOporace.Init` s identifikačním číslem klienta a je očekáváno kladné či záporné potvrzení: `paketOdServeru=mujProtokol.PrijmiTCPdata(TCPsocket)`. V případě kladného potvrzení jsou dále jsou vytvořena dvě vlákna, která slouží pro příjem unicastových a multicastových UDP zpráv zasílaných serverem.

Po úspěšně navázaném spojení může klient odesílat textové zprávy na server. Textová zpráva se napíše do příslušného místa ve formuláři. Pro její odeslání slouží metoda `button_odeslat_Click()`, která se vyvolá stiskem tlačítkem *Odeslat*, viz ukázka:

```
private void button_odeslat_Click(object sender, EventArgs e)
{
  TCPsocket.Blocking = true;//socket v blokovacím režimu
  TCPsocket.ReceiveTimeout = 10000;//limit čekání socketu na potvrzení
  string zprava = richTextBox2.Text+"\n";//načtení uživ. zadané zprávy
  if (zprava.StartsWith("*")){ operace = eOporace.bCastMessage; }
  else{ operace = eOporace.textMessage; }//jaký typ textové zprávy?
  //odeslání textové zprávy na server
  mujProtokol.PosliTCPdata(cisloKlienta, TCPsocket, operace, zprava);
  try
  {
    //čekání na potvrzení správného přijetí zprávy serverem
    PaketOdServeru = mujProtokol.PrijmiTCPdata(TCPsocket);
    if ((paketOdServeru != null) && (paketOdServeru.Operace ==
                                     eOporace.Acknowledge))
```

```
    { label_info.Text = "Server potvrzuje spravne prijeti posledne zaslane
                         zpravy"; }
    else
    { label_info.Text = "Server hlasi chybne prijeti posledne zaslane
                         zpravy"; }
  }
  catch (Exception)
  {
    label_info.Text = "Vyprsel casovy limit potvrzeni";
  }
}
```

V metodě `button_odeslat_Click()` se nejprve zkontroluje, zda se jedná o obyčejnou textovou zprávu určenou vlastnímu serveru nebo o zprávu začínající znakem "*" určenou i pro ostatní servery ve stejné lokální síti. Poté je tato zpráva odeslána metodou: `mujProtokol.PosliTCPdata(cisloKlienta, TCPsocket, operace, zprava)` a následně je očekáváno potvrzení o správném příjmu zprávy serverem. Časovač opět čeká na potvrzení 10s, jinak je zachycena chyba a předchozí textová zpráva by měla být uživatelem odeslána znovu.

Žádost o ukončení TCP spojení se serverem je vyvolána stiskem tlačítka *Odpojit*. Tuto žádost obstarává metoda `button_stop_Click()`, viz ukázka:

```
private void button_stop_Click(object sender, EventArgs e)
{
  TCPsocket.Blocking = true; //socket v blokovacím režimu
  operace = eOporace.End; //typ zprávy žádající ukončení TCP spojení
  string zprava = ""; //paket v tomto případě nenese žádnou text. zprávu
  try
  { //odeslání žádosti o ukončení spojení na server
    mujProtokol.PosliTCPdata(cisloKlienta, TCPsocket, operace, zprava);
    paketOdServeru = mujProtokol.PrijmiTCPdata(TCPsocket);
    if ((paketOdServeru != null) && (paketOdServeru.Operace ==
                                   eOporace.Ackonwledge))
    { //kladná odpověď, uzavření síťových prostředků a vláken
      label_info.Text = "Server akceptuje ukonceni spojeni";
      mujProtokol = null;
      if (multiUDPsocket != null) multiUDPsocket.Close();
      if (UDPsocket != null) UDPsocket.Close();
      PrijemUDPthread.Abort(); //ukončení
      PrijemTCPthread.Abort(); //vláken
    }
    else { label_info.Text = "Server neakceptuje ukonceni spojeni"; }
  }
  catch (Exception chyba){
  label_info.Text = "Potvrzeni o ukonceni spojeni neprislo v danem limitu";
  }
}
```

Typ odesílané zprávy musí být `eOporace.End` a tato zpráva je standardně odeslána metodou `PosliTCPdata()`. Poté je očekávána zpráva s potvrzením, že server akceptuje žádost o ukončení spojení. Časovač je opět nastavený na 10s. V případě kladného potvrzení jsou uzavřeny na straně klienta veškeré prostředky, které byly spojeny s vedením komunikace, a jsou také ukončena všechna vlákna. V opačném případě by žádost měla být odeslána znovu.

Textové zprávy, které odesílá server jen konkrétním klientům, jsou na straně klienta přijímány přes UDP socket, který je v blokovacím režimu. Příjem takovýchto zpráv zajišťuje metoda `CekejUDPZpravu()`, viz ukázka:

```
public void CekejUDPZpravu()
{
  byte[] UDBbbuffer = new byte[1006]; //místo pro příchozí data - bajty
  string prichoziText; //místo pro příchozí data - string
  UDPsocket = new Socket(AddressFamily.InterNetwork, SocketType.Dgram,
                        ProtocolType.Udp); //vytvoření UDP socketu
  IPEndPoint UDPhost = new IPEndPoint(IPAddress.Any, 8000);
  UDPsocket.Bind(UDPhost); //svázání socketu s lokální IP a portem
  try
  {
    while (true)
    {
      UDPsocket.Receive(UDBbbuffer); //příjem dat
      prichoziText = Encoding.ASCII.GetString(UDBbbuffer); //string na byte
      if (prichoziText.StartsWith(Prefix)) //prefix by měl být 0xF0F0
      {
        prichoziText = prichoziText.Remove(0, 6); //odstranění prefixu
        ZobrazZpravu(prichoziText); //zobrazení zprávy
      }
    }
  }
  catch (Exception){ UDPsocket.Close(); return; }
}
```

UDP socket je svázán s lokální IP adresou a portem metodou `UDPsocket.Bind(UDPhost)` a dále se v nekonečné smyčce čeká v blokovacím režimu socketu na příchozí zprávu, a to zajišťuje metoda socketu `UDPsocket.Receive(UDBbbuffer)`. Poté je provedena kontrola, zda příchozí zpráva obsahuje prefix 0xF0F0, a tedy zda je určena pro tuto aplikaci.

Textové zprávy určené pro všechny klienty daného serveru jsou serverem zasílány na adresu multicastové skupiny, jejímiž členy jsou právě všichni klienti tohoto serveru. Příjem takovýchto zpráv zajišťuje metoda `cekejMultiZpravu()`, viz ukázka:

```
public void cekejMultiZpravu()
{
  byte[] UDBbbuffer = new byte[1006]; //místo pro příchozí data - bajty
  string prichoziText; //místo pro příchozí data - string
  multiUDPsocket = new Socket(AddressFamily.InterNetwork, SocketType.Dgram,
                              ProtocolType.Udp);//inicializace UDP socketu
  IPEndPoint IPeP = new IPEndPoint(IPAddress.Any, 7000);

  multiUDPsocket.Bind(IPeP); //svázání socketu s lokální IP a portem
  IPAddress ip = IPAddress.Parse("224.5.6.7");//IP adresa multic. Skupiny
  //přidání socketu do multicastové skupiny
  multiUDPsocket.SetSocketOption(SocketOptionLevel.IP,
                                 SocketOptionName.AddMembership,
                                 new MulticastOption(ip, IPAddress.Any));
  try
  {
    while (true)
    {
```

```
            multiUDPsocket.Receive(UDBbbuffer); //příjem multicastové zprávy
            prichoziText = Encoding.ASCII.GetString(UDBbbuffer); //string na byte
            if (prichoziText.StartsWith(Prefix)) //měl by být 0xF0F0
            {
              prichoziText = prichoziText.Remove(0, 6); //odstranění prefixu
              ZobrazZpravu(prichoziText); //zobrazení zprávy
            }
        }
    }
    catch (Exception){ multiUDPsocket.Close(); return; }
}
```

UDP socket je svázán s lokální IP adresou a portem metodou `multiUDPsocket.Bind(UDPhost)` a dále je tento socket přidán do multicastové skupiny přes metodu `multiUDPsocket.SetSocketOption()`. Poté se v nekonečné smyčce čeká v blokovacím režimu socketu na příchozí zprávu, a to zajišťuje metoda socketu `multiUDPsocket.Receive(UDBbbuffer)`. Nakonec je provedena kontrola, zda příchozí zpráva obsahuje prefix 0xF0F0, a tedy zda je určena pro tuto aplikaci.

7 Doplňkové studie k vypracování programu

Během programování projektu MPKT bylo nutné vypořádat se se situacemi, které měly poněkud složitější řešení. Následuje jejich stručný popis a nástin možného řešení v jazyce C#.

7.1 Převod struktury/třídy na proud bajtů

Jednou z takových situací je převedení paketu MPKT, který je datového typu struktura, na prostý proud bajtů připravený k odesláním pomocí socketu. V jazyce C/C++ je vhodné použít funkci `memcpy()`, která překopíruje blok paměti, tedy určitý počet bajtů, do připraveného pole datového typu byte, a ten je možné přes socket odeslat po síti.
Následuje příklad řešený v jazyce C++:

```
struct Paket //struktura představující paket
{
   unsigned short id;
   unsigned short delka;
   char operace;
   unsigned short CRC16;
   char zprava[1000];
};

Paket *dPaket = new Paket; //dynamicky vytvořená struktura typu Paket
char proudB[sizeof(Paket)]; //vytvoření pole bajtů
//překopírování bloku paměti do pole bajtů
memcpy(proudB, dPaket, sizeof(Paket));
//odeslání struktury paketu ve formě proudu bajtů
send(mujSocket, proudB, sizeof(proudB), 0);

//přijetí struktury paketu ve formě proudu bajtů
recv(mujSocket, proudB, sizeof(proudB), 0);
//překopírování bloku paměti do datového typu struktura
memcpy(dPaket, proudB, sizeof(proudB));
```

V jazyce C# je vhodné využít metodu ***serializace***. Ta slouží k převzetí objetku v paměti a jeho převedení na proud bajtů, který lze uložit nebo odvysílat. Opačným způsobem pak pracuje deserializace, která přebírá proud bajtů a vytváří z něj instanci třídy v paměti. Případ s převedením objektu na proud bajtů může v jazyce C# vypadat následovně:

```
[Serializable]      //nastavení podpory serializace
Public class Paket //definice třídy Paket, jen příklad
{
   public ushort id;
   public ushort delka;
   public char operace;
   public ushort CRC16;
   public char zprava[512];
}
```

Do hlavní části programu se umístí následující kód pro převod instance třídy na proud bajtů:

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

```
Paket mujPaket = new Paket();
byte[] poleBajtu = new byte[1024]; //buffer pro odesílaná data
//nad objekty typu Stream  lze provádět serializaci
MemoryStream mujStream = new MemoryStream();
IFormatter lFormatter = new BinaryFormatter();

//serializace objektu Paket do objektu typu Stream
lFormatter.Serialize(mujStream, mujPaket);
poleBajtu = mujStream.ToArray(); //převod objektu typu Stream na pole bajtů
mujSocket.Send(poleBajtu); //odeslání objektu jako pole bajtů přes socket

mujStream.Flush(); //vymaže obsah mujStream
```

Sestavení objektu Paket z příchozího proudu bajtů na straně příjemce:

```
byte[] poleBajtu = new byte[1024]; //buffer pro přijímaná data
//nad objekty typu Stream  lze provádět deserializaci
MemoryStream mujStream = new MemoryStream();

mujSocket.Receive(poleBajtu); //příjem dat přes socket
//zápis pole bajtů do objektu typu Stream
mujStream.Write(poleBajtu, 0, poleBajtu.Length);

Paket mujPaket = null; //prostor pro deserializovaný objekt
IFormatter lFormatter = new BinaryFormatter();
mujStream.Position = 0; //nastavení pozice na počátek ve streamu
mujPaket = (Paket)lFormatter.Deserialize(mujStream);//deserializace

mujStream.Flush(); //vymaže obsah mujStream
```

Výhodou serializace je, že je bezpečná, co se týče přístupu do paměti. Její nevýhodou v našem případě je, že se spolu s daty námi definovaného paketu také přenáší potřebné režijní informace nutné k zpětnému sestavení objektu z proudu bajtů – deserializaci.

Druhým možným způsobem, jak převádět struktury na proud bajtů je použití neřízené, tedy typově a paměťově nezabezpečené, třídy **Marshal**. Tato třída nabízí mimo jiné prostředky ukazatele na část paměti a kopírování bloků paměti. V následující ukázce je uvedeno překopírování struktury z paměti do pole bajtů a zpětný převod tohoto pole bajtů na strukturu:

```
using System.Runtime.InteropServices; //zpřístupní třídu Marshal
Public struct Paket //definice struktury Paket, jen příklad
{
   public ushort id;
   public ushort delka;
   public char operace;
   public ushort CRC16;
   public char zprava[512];
}
Paket paket = new Paket(); //struktura má nyný všechny položky vynulovány

byte[] poleBajtu = new byte[Marshal.SizeOf(paket)]; //odchozí data
//vytvoření ukazatele na strukturu
IntPtr ukazatel = Marshal.AllocHGlobal(Marshal.SizeOf(paket));
//svázání ukazatele s místem v paměti, kde se nachází struktura
Marshal.StructureToPtr(paket, ukazatel, false);
//překopírování bloku paměti, kam ukazuje ukazatel, do pole bajtů
Marshal.Copy(ukazatel, poleBajtu, 0, Marshal.SizeOf(paket));
//toto pole bajtů je nyní možno standardně odeslat přes socket
```

```
Paket paket2 = new Paket(); //nová struktura pro příjmaná data
//vytvoření ukazatele na novou strukturu
IntPtr ukazatel2 = Marshal.AllocHGlobal(Marshal.SizeOf(paket2));
//překopírování pole bajtů na místo do paměti, kam ukazuje ukazatel
Marshal.Copy(poleBajtu, 0, ukazatel2, Marshal.SizeOf(paket2));
//naplnění struktury daty, kam ukazuje ukazatel
paket2 = (Paket)Marshal.PtrToStructure(ukazatel, typeof(Paket));
```

7.2 Dynamický seznam klientů

Je potřeba, aby si server vedl seznam svých připojených klientů. Protože však dopředu není známo, kolik klientů bude v určitém časovém intervalu server obsluhovat, bylo výhodné zvolit dynamický jednosměrně vázaný seznam. Takový typ seznamu nabízí možnosti jako odstraňovat nebo vkládat prvky z/do jakéhokoliv místa tohoto seznamu. Každý prvek obsahuje referenci na prvek následující, kromě prvku posledního. V projektu MPKT verze 2008 byl využit předpřipravený programový kód v jazyce C/C++ realizující výše zmíněný dynamický seznam. Více viz literatura [1].

V jazyce C# je možno pro potřebu realizace dynamického seznamu využít přímo implementovanou třídu `LinkedList` [2]. Následuje popis této třídy:

Tab.7.1: Atributy třídy LinkedList

Atributy	
Count	aktuální počet prvků seznamu
First	reference na první prvek seznamu
Last	reference na poslední prvek seznamu

Tab. 7.2: Metody třídy LinkedList

Metody	
AddFirst()	přidá nový prvek na první místo v seznamu
AddLast()	přidá nový prvek za poslední prvek v seznamu
AddAfter()	přidá nový prvek za aktuální prvek
AddBefore()	přidá nový prvek před aktuální prvek
Clear()	odebere všechny prvky ze seznamu
Remove()	odebere aktuální prvek ze seznamu
Find()	najde první prvek, který odpovídá hledané hodnotě

7.3 Zašifrování přenášených zpráv

Jazyk C# obsahuje třídy pro práci se všemi elementy modelu Code Access Security rámce .NET, včetně zásad zabezpečení, sad oprávnění a důkazů. Tyto třídy jsou k dispozici ve jmenném prostoru `System.Security.Cryptography` a podporují např. šifrovací algoritmy jako DES, 3DES, RSA, Rijndael, MD5.

Na následující straně je uveden příklad použití symetrické šifry Rijndael:

```
static void Main(string[] args)
{
  //****************ŠIFROVÁNÍ**********************
  RijndaelManaged mujRijndael = new RijndaelManaged();//instance Rijndael
  byte[] key = mujRijndael.Key;//šifrovací klíč
  byte[] iVector = mujRijndael.IV;//inicializační vektor
  //vytvoření šifrátoru
  ICryptoTransform encryptor = mujRijndael.CreateEncryptor(key, iVector);

  MemoryStream Mstream = new MemoryStream();//stream pro uložení zaš. textu
  //stream, který propojí šifrování s uložením do Mstream
  CryptoStream Cstream = new CryptoStream(Mstream, encryptor,
                                        CryptoStreamMode.Write);
  using (StreamWriter swEncrypt = new StreamWriter(Cstream))
  { //zápisem textu do Cstream se odstartuje šifrování
    swEncrypt.Write("Tento text bude zasifrovan");
  } //převod zašifovaného textu(streamu) na bajty
  byte[] zasifrovanyText = Mstream.ToArray();

  //****************DEŠIFROVÁNÍ*********************
  string desifrovanyText;//místo pro dešifrovaný text
  //vytvoření dešifrátoru
  ICryptoTransform decryptor = mujRijndael.CreateDecryptor(key, iVector);
  //stream, kde je zašifrovaný text
  MemoryStream msDecrypt = new MemoryStream(zasifrovanyText);
  //stream, do kterého se načte a dešifruje zašifrovaný text
  CryptoStream csDecrypt = new CryptoStream(msDecrypt, decryptor,
                                          CryptoStreamMode.Read);
  StreamReader srDecrypt = new StreamReader(csDecrypt);
  //uložení dešifrovaného textu jako string
  desifrovanyText = srDecrypt.ReadToEnd(); }
```

Příklad použití asymetrické šifry RSA:

```
static void Main(string[] args)
{
  //****************ŠIFROVÁNÍ**********************
  //instance RSA, klíč délky 384
  RSACryptoServiceProvider mojeRSA = new RSACryptoServiceProvider(384);
  //export parametrů objektu mojeRSA
  RSAParameters RSAKeyInfo = mojeRSA.ExportParameters(false);
  //načtení veřejného klíče
  byte[] verejnyKlic = RSAKeyInfo.Modulus;

  string text = "Tento text bude zasifrovan";
  byte[] bText = encoding.GetBytes(text); //převod řetězce na bajty
  //provedeno zašifrování veřejným klíčem
  byte[] zasifrovanyText = mojeRSA.Encrypt(bText, false);

  //****************DEŠIFROVÁNÍ*********************
  byte[] desifrovanyText = mojeRSA.Decrypt(zasifrovanyText, false);
  text = encoding.GetString(desifrovanyText);//dešifrovaný text
}
```

Výše uvedené postupy symetrického a asymetrického šifrování by mohly být zahrnuty do komunikačního protokolu. Server by v potvrzovací zprávě o akceptování připojení klienta zaslal také svůj veřejný šifrovací klíč. Klient by následně vygeneroval symetrický šifrovací klíč, zašifroval by ho veřejným klíčem serveru a odeslal. Poté by již mohla probíhat symetricky šifrovaná komunikace mezi klientem a serverem.

Závěr

V první části této práce byl představen původní komunikační protokol MPKT 2008 sloužící pro komunikaci typu klient-server, kterým se v rámci výuky demonstrují základní principy práce komunikačního protokolu. Smyslem protokolu MPKT je nadstandardní zabezpečení přenosu textových zpráv, které jsou přenášeny pomocí transportních protokolů TCP a UDP. Byla provedena jeho analýza a navrženy možnosti rozšíření tohoto protokolu o další položky jeho hlavičky, přibližující jej v praxi používaným protokolům. Dále byla představena platforma .NET, která je určena jak pro provoz rozsáhlých distribuovaných aplikací prostřednictvím sítě internet, tak pro klasické lokální aplikace. Na této platformě byly vytvořeny dvě softwarové aplikace. Jedna vystupuje jako klient a druhá rozšířená o patřičné možnosti a ovládací prvky vystupuje jako server. Tyto aplikace spolu vzájemně komunikují prostřednictvím rozšířeného protokolu MPKT. Tento protokol je samozřejmě možné dále rozšiřovat a přibližovat jej reálným protokolům, avšak to by již přesahovalo rámec této práce. Pro implementaci tohoto protokolu byl použit programovací jazyk C#, který je velmi úzce spjatý s platformou .NET. Oproti jazyku C++, ve kterém byl implementován původní protokol MPKT, je C# silně typově zabezpečený jazyk, který odstraňuje celou kategorii chyb v programech z jazyků C a C++ vyplývajících z neplatných převodů a nesprávné aritmetiky ukazatelů. Dále disponuje velmi výkonným mechanismem uvolňování již nepotřebné operační paměti. V rámci komunikace s protokolem MPKT mezi vytvořenými aplikacemi je klíčová příprava dat k odeslání přes socket formou proudu bajtů. V jazyku C++ toto bylo řešenou formou překopírování bloku paměti, představující paket, a ten mohl být poté odeslán do sítě. V jazyku C# je možno použít obdobný postup využívající kopírování bloku paměti a ukazatel. Tento přístup se však v rámci vytvořeného programu ukazoval jako náchylný k chybám a aplikace se nechovala korektně. Proto byla na převedení struktury/třídy na proud bajtů zvolena, co se týče přístupu do paměti bezpečnější, technika serializace objektu a k ní inverzní deserializace. Dále byly v rámci jazyka C# zmíněny způsoby šifrování pomocí symetrického a asymetrického šifrovacího algoritmu, které by mohly být také vhodně začleněny do protokolu MPKT. Jazyk C# se z programátorského hlediska ukázal jako velmi srozumitelný programovací jazyk vysoké úrovně a v kombinaci s kvalitním vývojovým prostředím je vhodné začlenit jej do výuky. Právě pro možnosti jeho nasazení do výuky byly dále v rámci této práce z části nově vytvořeny a z části aktualizovány výukové texty pro předmět Pokročilé komunikační techniky.

Literatura

[1] Malý, J.: *Implementace komunikačního protokolu MPKT v C++*, VUT v Brně, 2008.

[2] Drayton P., Albahari B., Neward T.: *C# v kostce*, GRADA Publishing, 2003.

[3] Puš, P.: *Poznáváme C# a Microsoft .NET* [online]. Dostupné z URL < http://www.zive.cz/Clanky/Poznavame-C-a-Microsoft-NET--1dil/sc-3-a-120978/default.aspx>.

[4] Chappell,D.: *COM and CORBA* [online]. Dostupné z URL <http://www.chappellassoc.com/articles/article_COM_and_CORBA.html>.

[5] *RSA* [online]. Dostupné z URL <http://cs.wikipedia.org/wiki/RSA>.

[6] Wiszczor, T.: *.NET Remoting pod lupou* [online]. Dostupné z URL <objekty.pef.czu.cz/2003/sbornik/Wiszczor2003.pdf>.

[7] Dvorek, L.: *Projekt Mono* [online]. Dostupné z URL <http://www.owebu.cz/programovani/vypis.php?clanek=1662>.

Seznam příloh

A Obsah přiloženého CD……………………………………………………………………53
B Odeslání paketu včetně fragmentace…………………………………………………...54
C Přijetí paketu včetně defragmentace…………………………………………………....55
D Základní orientace v programování v jazyce C#...56

A Obsah přiloženého CD

V rámci této práce byly také provedeny aktualizace výukových textů pro potřeby programovacího jazyka C#. Celkem byly vytvořeny 4 texty:

- **Základy programování v jazyce C#** - rozsah 13 stran, také uveden v příloze D
- **Sockety v C# -** rozsah 8 stran
- **Vlákna v C#** - rozsah 5 stran
- **Doplňkové studie k projektu MPKT** - rozsah 5 stran

Části těchto textů jsou také zahrnuty v samotném textu této diplomové práce.

Dále byly v rámci této práce vytvořeny dvě softwarové aplikace pro práci v síťovém prostředí, které spolu komunikují pomocí rozšířeného protokolu MPKT. K programování byl použit programovací jazyk C#. Pro chod těchto aplikací je zapotřebí mít na stanicích nainstalovaný .NET Framework 3.5 Testováno na OS Windows XP64 SP2.

Pokyny:
Obě aplikace je možno spustit na jedné pracovní stanici. V tomto případě spolu budou komunikovat přes lokální smyčku.

Server – po spuštění aplikace možno změnit číslo portu, na kterém bude server naslouchat příchozím požadavkům na TCP spojení. Je však vhodné ponechat původní hodnotu. Poté spustit tlačítkem *Start* naslouchání serveru. Připojí-li se nějaký klient je o tom server patřičně informován a klientova IP adresa je zobrazena v seznamu připojených klientů. Ze serveru lze směrem k jeho klientům zasílat textové zprávy. Pro zaslání zprávy všem připojeným klientům je nutné mít zaškrtnuté políčko *Vsem klientum*. Pro zaslání zprávy určité skupině klientů je nutno tuto skupinu prvně stanovit. Kliknutím na IP požadovaného klient a stiskem tlačítka *Pridat* se jeho IP adresa přidá do seznamu *Skupina.* Toto opakujeme pro požadované klienty. Poté je nutno zaškrtnout políčko *Skupina* a stiskem tlačítka *Odeslat* se zadaná zpráva odešle této supině klientů. Naslouchání serveru se zastaví stiskem tlačítka *Stop*.

Klient – po spuštění aplikace je nutno zadat IP adresu serveru, případně změnit číslo portu, na kterém server naslouchá příchozím požadavkům na TCP spojení. Pro komunikaci v rámci jedné stanice se zadá IP adresa **127.0.0.1**. Dále je nutno zadat identifikační číslo, které musí spadat do rozsahu 0-255. Poté je možno vyslat požadavek na spojení se serverem stiskem tlačítka *Pripojit*. Klient je informován o akceptování či neakceptování žádosti o spojení se serverem. V případě neakceptování spojení musí klient zvolit jiné identifikační číslo a pokusit se připojit znovu. Poté již může zadávat textové zprávy do formuláře *Odchozi zprava* a stiskem tlačítka *Odeslat* je zpráva odeslána na server.

Zdrojové soubory: 1) Klient: Program\Klient\WindowsFormsApplication2\KlientAplikace.cs
2) Server: Program\Server\WindowsFormsApplication1\ServerAplikace.cs

B Odeslání paketu včetně fragmentace

```
public void PosliTCPdata(byte id, Socket klient, eOporace operace, string zprava)
{
  ushort delkaZpravy = (ushort)zprava.Length;
  Crc16 mojeCRC16 = new Crc16();
  ushort crc;
  Paket mujPaket = new Paket(); //instance pro nový paket
  byte[] classStream = new byte[1024];
  //objekty nutné pro serializaci
  MemoryStream mujStream = new MemoryStream();
  IFormatter lFormatter = new BinaryFormatter();
  //inicializace položek, které řídí fragmentaci
  string fragment;
  ushort maxDelka = 100;
  ushort delkaFragmentu = 100;
  ushort pocetFragmentu = 1;
  ushort posledniFragment = 0;
  ushort pocatekFragmentu = 0;
  bool mf = false;
  try
  {
    posledniFragment = (ushort)(delkaZpravy % delkaFragmentu); //délka posledního fragmentu
    if (delkaZpravy > delkaFragmentu)
    { //zjištění počtu fragmentů zprávy
      pocetFragmentu = (ushort)(delkaZpravy / delkaFragmentu);
      mf = true;
      if (posledniFragment != 0) pocetFragmentu++;
    }
    else if (delkaZpravy != delkaFragmentu)
    {
      delkaFragmentu = posledniFragment;
    }
      mujPaket.CisloKlienta = id;
      mujPaket.Operace = operace;

      for (ushort i = 1; i <= pocetFragmentu; i++) //postupně se odešlou všechny fragmenty
      {
        fragment = zprava.Substring(pocatekFragmentu, delkaFragmentu); //vytvoření fragmentu
        if (i == pocetFragmentu) mf = false; //nastavení položky MoreFragments na true X false
        if ((operace == eOporace.textMessage) || (operace == eOporace.bCastMessage))
        { //sestavení paketu s textovou zprávou
          crc = mojeCRC16.Checksum(fragment);
          mujPaket.Offset = (ushort)(i * maxDelka
          mujPaket.MoreFragments = mf;
          mujPaket.DelkaZpravy = delkaFragmentu;
          mujPaket.CRC16 = crc;
          mujPaket.Zprava = fragment;
        }
        else //jedná se o netextovou zprávu
        { //sestavení paketu
          mujPaket.Offset = 0;
          mujPaket.MoreFragments = false;
          mujPaket.DelkaZpravy = 0;
          mujPaket.CRC16 = 0;
          mujPaket.Zprava = zprava;
        }
         //následuje proces serializace paketu na proud bajtů
        mujStream.Position = 0;
        lFormatter.Serialize(mujStream, mujPaket);
        classStream = mujStream.ToArray();
        klient.Send(classStream); //odeslání proudu bajtů přes socket příjemci

        if ((i == pocetFragmentu - 1) && (posledniFragment != 0))
        {  delkaFragmentu = posledniFragment; }

        pocatekFragmentu += maxDelka; //nové nastavení offsetu
        mujStream.Flush();  //vyprázdnění streamu
      }
    }
  catch (Exception) { return; }
  finally { mujStream.Close(); }
}
```

C Přijetí paketu včetně defragmentace

```
public Paket PrijmiTCPdata(Socket klient)
{
  int i = 0;
  ushort PocetFragmentu = 0;
  Paket[] polePaketu;
  //objekty nutné pro deserializaci
  byte[] classStream = new byte[1024];
  MemoryStream mujStream = new MemoryStream();
  IFormatter lFormatter = new BinaryFormatter();
  Paket mujPaket = null;  //paket pro naplnění příchozími daty
  try
  {
    klient.Receive(classStream);  //příjem proudu bajtů
    mujStream.Write(classStream, 0, classStream.Length); převod proudu bajtů na typ stream

    mujStream.Position = 0; //pozice ve streamu na počátek
    mujPaket = (Paket)lFormatter.Deserialize(mujStream); //deserializace proudu bajtů
    mujStream.Flush(); //uvolnění streamu

    if ((mujPaket.Operace == eOporace.textMessage) || (mujPaket.Operace ==
         eOporace.bCastMessage)) //zda textová zpráva?
    {
      if (mujPaket.MoreFragments == true) //je zpráva rozdělena na více fragmentů?
      {
        StringBuilder CelaZprava = new StringBuilder();
        polePaketu = new Paket[25]; //vyrovnávací paměť pro příchozí fragmenty
        polePaketu[i] = mujPaket;
        CelaZprava.Append(mujPaket.Zprava); //připojení fragmentu do zprávy
        do
        { //příjem a deserializace fragmentů
          MemoryStream pomocnyStream = new MemoryStream();
          klient.Receive(classStream);
          pomocnyStream.Write(classStream, 0, classStream.Length);

          pomocnyStream.Position = 0; //pozice ve streamu na počátek
          mujPaket = (Paket)lFormatter.Deserialize(pomocnyStream);

          i++;
          polePaketu[i] = mujPaket; //fragment do vyrovnávací paměti
          pomocnyStream.Flush(); //vyprázdnění a uzavření proudu
          pomocnyStream.Close();

          CelaZprava.Append(mujPaket.Zprava); //připojení fragmentu do zprávy
          PocetFragmentu++;
        }
        while(mujPaket.MoreFragments != false);
        //provádění kontroly délky fragmentu a CRC
        for (i = 0; i < PocetFragmentu; i++)
        {
          if (!KontrolaZpravy(polePaketu[i]))
          return null;
        }
        mujPaket.Zprava = CelaZprava.ToString();
        return mujPaket;
      }
      else //zpráva nebyla fragmentována
      {
        if (KontrolaZpravy(mujPaket))
          return mujPaket;
        else return null;
      }
    }
    else return mujPaket;
  }
  catch (Exception)
  {
    return null;
  }
  finally {  mujStream.Close();  } //uzavření streamu
}
```

D Základní orientace v programování v jazyce C#

1 Základy programování v jazyce C#

Základním požadavkem pro běh C# aplikací je nutné mít nainstalované běhové prostředí .NET framework. Dále čerpáno z literatury [2], [3]. Ukázka jednoduchého programu:

```
using System;
namespace ukazky_kodu
{
   class AhojSvete
   {
      public static void Main(string[] args)
      {
         Console.WriteLine("Ahoj svete!");
         Console.ReadLine();
      }
   }
}
```

Na prvním řádku se příkazem **using** importuje knihovna System, která obsahuje třídu Console, jejíž metody budou použity. Řádek s klíčovým slovem **namespace** určuje, do kterého jmenného prostoru třída patří (v našem případě patří do jmenného prostoru ukazky_kodu). Jmenné prostory slouží v .NETu k oddělení tříd do k sobě logicky patřících částí. Řádek s klíčovým slovem **class** určuje název vytvářené třídy (AhojSvete). Po něm následuje řádek definující metodu **Main** s identifikátory přístupu public a static, která nevrací žádnou hodnotu, což je dáno slovíčkem **void**. Do metody jako parametr vstupuje pole řetězců (string [] args). Všechny tyto podmínky musí metoda splňovat, aby třída AhojSvete byla spustitelná konzolová aplikace. Příkaz Console.WriteLine vypíše předaný parametr (v našem případě je to řetězec „Ahoj svete!“) na obrazovku. Jednotlivé bloky programu jsou uzavřeny do složených závorek.

Jazyk C# je objektově orientovaný jazyk vycházející z programovacích jazyků Java a C++. Stejně jako tyto jazyky je i C# **case-sensitive**, což znamená, že významově odlišuje velká a malá písmena.V tomto jazyce je realizováno 80% základních knihoven .NET Frameworku. I přesto, že je koncipován hlavně pro psaní řízeného kódu, na jehož užití je platforma .NET postavena, lze jej v případě potřeby využít i pro tvorbu kódu neřízeného (bloky unsafe). Použití neřízeného kódu znamená, že běhové prostředí CLR neověřuje, zda-li je napsaný kód bezpečný (například se neověřuje jinak vyžadovaná typová bezpečnost).

1.1 Základní prvky jazyka C#:

- Třídy – základní stavební prvek při tvorbě objektově orientovaných aplikací obsahující akce (metody) a atributy.
- Struktury – lze je chápat jako zjednodušené třídy, jejich užitím jsou nejčastěji popisovány vlastní datové struktury.
- Výčtové typy
- Vlastnosti – někdy označované jako chytré proměnné.

- Pole a jejich „chytrá“ verze nazývaná indexery .
- Zástupci – typově bezpečné ukazatele na funkce.
- Události – druh zástupců sloužící ke zpracování asynchronních operací.

1.2 Objektové programování v jazyku C#

Objektově orientovaný přístup k vývoji softwaru je v dnešní době velmi využívaný. Důvody použití tohoto přístupu na rozdíl od přístupu strukturovaného, používaného například v jazyku C, mimo jiné jsou:

- přehlednější zdrojové kódy,
- vyšší znovu použitelnost vytvořených aplikací,
- jednodušší správa aplikací.

Objekty a třídy

Objekty ve světě vývoje aplikací často představují abstrakci objektů reálného světa. To s sebou přináší, větší možnosti při návrhu struktury jednotlivých částí aplikace. Každý objekt je **instancí** neboli výskytem **třídy**. Třídu lze chápat jako šablonu pro vytváření objektů.
Každá třída může obsahovat definici pro:

- **metody –** operace, které může třída nebo pomocí ní vytvořený objekt provádět
- **atributy –** představují stavy/data třídy nebo vytvořeného objektu

Zapouzdření objektů

Zapouzdření je jednou ze základních vlastností objektově orientovaného přístupu a existuje z důvodu, že data a operace objektu se vzájemně ovlivňují – tvoří nedělitelný celek. Zapouzdření znamená, že objekt má některé své členy (metody/atributy) před okolím skryty. Členy přístupné okolí se nazývají **rozhraní** objektu.

Se zapouzdřením souvisí i vlastnosti nazývané:

- **selektory** – slouží k získání hodnoty nějakého skrytého atributu, ale neumožňují tuto hodnotu modifikovat,
- **modifikátory** – slouží ke změně hodnoty nějakého skrytého atributu.

Konstruktor

Každá třída musí mít konstruktor, který představuje předpis na vytvoření nového objektu. Konstruktor nejčastěji obsahuje kód pro inicializaci (první nastavení hodnoty) některých, nebo všech atributů nově vznikajícího objektu. Pokud ho ve své třídě neuvedeme, použije se implicitní (výchozí) konstruktor. Zápis definice konstruktoru je následující:

```
public Nazev_Tridy()
{
/*kód operací, které se provedou vždy při vytváření nového objektu podle
této třídy*/
}
```

Konstruktoru mohou být také předány parametry, se kterými bude při vytváření nového objektu pracovat. Definice konstruktoru očekávajícího parametry se zapisuje takto:

```
public Nazev_Tridy(datovy_typ Nazev_parametru, datovy_typ
                   Název_parametru2)
{
/*kód operací, které se provedou vždy při vytváření nového objektu podle
této třídy a využívají předané parametry*/
}
```

Atributy a vlastnosti

Atributy třídy se v C# definují zápisem:

```
Specifikator_pristupu datovy_typ Nazev_Atributu;
```

Vlastnosti určují, co se provede za akce při čtení atributu (blok **get**) a při jeho modifikaci (blok **set**). Jejich zápis je ve tvaru:

```
Specifikator_pristupu datovy_typ Nazev_Vlastnosti
{
   get
   {
      //získání hodnoty privátního atributu
      return výsledná_hodnota;
   }
   set
   {
      //nastavení hodnoty privátního atributu
      název_atributu = value;
   }
}
```

Získání hodnoty se následně provádí zápisem:

```
nazev_promenne = Nazev_Objektu.Nazev_Vlastnosti;
```

Nastavení hodnoty pak:

```
Nazev_Objektu.Nazev_Vlastnosti = hodnota;
```

Metody

Operace, které se mohou nad daným objektem vykonávat, se nazývají metody objektu. Zápis metody vypadá následovně:

```
Specifikator_pristupu datovy_typ Nazev_metody()
{
   //kód metody
   return návratová_hodnota;
}
```

V případě, že je jako datový typ metody uveden **void**, metoda žádnou hodnotu nevrací a slovíčko **return** není poviné. Také je možné, aby byli metodě při volání předány nějaké parametry. Zápis metody by v tomto případě byl následovný:

```
Specifikator_pristupu datovy_typ Nazev_metody(datovy_typ
                           Nazev_parametru, datovy_typ Nazev_parametru2)
{
   //kód metody
   return návratová_hodnota;
}
```

Specifikátory přístupu

Pro určení viditelnosti členů vně třídy nebo vytvořeného objektu slouží specifikátory přístupu, které jsou v C# představovány klíčovými slovy:

- **private** – specifikuje, že člen je privátní (soukromý) - přístupný pouze uvnitř třídy,
- **public** – specifikuje, že člen je veřejný – přístupný z jiných tříd,
- **protected** – specifikuje, že člen je přístupný pouze potomkům třídy.

Dědičnost v C#

Pojem dědičnost v oblasti objektově orientovaného programování znamená možnost odvodit nějakou třídu z třídy jiné. Potřeba využít dědičnosti nastává tehdy, je-li jedna třída speciálním případem jiné třídy. Třída, která slouží jako základ pro odvozenou třídu nese označení bázová třída. V jazyku C# může být pro každou třídu určen pouze jeden předek – jednoduchá dědičnost. To je rozdíl oproti jazyku C++, kde existovala několikanásobná dědičnost. Nepřítomnost několikanásobné dědičnosti řeší jazyk C# stejně jako Java použitím rozhraní.

Zápis definice třídy, která dědí od jiné třídy se zapisuje v C# následujícím způsobem:

```
public class NazevOdvozeneTridy : NazevBazoveTridy
{
   //implementace třídy
}
```

Zápis konstruktoru, který volá konstruktor implementovaný v bázové třídě:

```
public NazevOdvozeneTridy() : base()
{
   //implemetace dodatečných operací
}
```

Pokud v metodě, která je implementována, potřebujeme zavolat metodu implementovanou na bázové třídě, opět využijeme klíčového slova **base** a to tímto způsobem:

```
public NazevMetody()
{
   base.NazevBazoveMetody();
}
```

S dědičností souvisí specifikátor přístupu **protected**. Tento specifikátor přístupu použijeme v případě, kdy chceme, aby daný člen byl přístupný pouze odvozeným třídám. S implementací dědičnosti se často setkáváme s potřebou předefinovat funkčnost nějaké metody v odvozené třídě. Tomuto předefinování se také říká **překrytí**. V bázové třídě musí být tato metoda definována jako **virtuální**. K překrývání metod v odvozených třídách slouží

klíčová slova **override** a **new**. Override použijeme v případě, kdy nová metoda v odvozené třídě plní logicky stejnou funkci , má stejný specifikátor přístupu a stejný návratový typ jako metoda v bázové třídě. Slovíčko **new** použijeme v opačném případě, když nová metoda plní logicky jinou funkci nebo mění specifikátor přístupu a nebo má jiný návratový typ.

Přetěžování metod a konstruktorů

Metody a konstruktory mohou být v rámci třídy definovány v několika verzích, do kterých vstupuje rozdílný počet parametrů. Takovéto metody/konstruktory jsou nazývány **přetížené**. V případě metod musejí mít všechny verze stejný návratový typ. Příkladem je metoda `Secti()`. Po přetížení bude možno při volání této metody použít dva nebo tři vstupní parametry:

```
public int Secti(int a, int b)
{
   int vysledek = a + b;
   return vysledek;
}

public int Secti(int a, int b, int c)
{
   int vysledek = a + b + c;
   return vysledek;
}
```

Statické členy tříd

Členy jsou nazývány **instanční**, pokud jsou svázány až s konkrétním objektem. Existují však ještě členy třídy, které využijeme v případě, když potřebujeme, aby nebyly spojeny s nějakou konkrétní instancí dané třídy, ale s třídou jako takovou. Takovéto členy označujeme jako členy **statické**. K těmto členům potom přistupujeme nikoliv prostřednictvím jména instance, ale prostřednictvím jména třídy. Statické mohou být jak atributy tak metody třídy.

Polymorfismus

Polymorfismus, neboli **mnohotvárnost**, je jeden z důležitých aspektů v objektově orientovaném programování. Polymorfismus znamená možnost použít mnoho druhů typů se stejným rozhraním (veřejně přístupnými operacemi a atributy) bez ohledu na detaily jejich implementace. Polymorfismus tedy zajišťuje to, že stačí znát rozhraní bázové třídy a použitím tohoto rozhraní volat operace a atributy, které jsou různě naimplementovány v odvozených třídách. Polymorfismus je tedy také svázán s použitím dědičnosti.

Abstraktní třída a rozhraní

Abstraktní třídy se používají v situaci, kdy je potřeba definovat "dohodu" o tom, jaké členy bude odvozená třída muset implementovat. Abstraktní třídy se chovají stejně jako normální třídy až na to, že mají jednu nebo více členských metod definované jako abstraktní. Abstraktní metody jsou reprezentovány pouze hlavičkou funkce, tedy definicí jejího návratového typu, počtu vstupních parametrů a jejich typů.

```
//definice abstraktní třídy
public abstract class AbstaktniTrida
{
   //tato metoda musí být v odvozené třídě naimplementována
   public abstract void AbstraktniMetoda();

   public void NormalniMetoda()
   {
      //implementace metody
   }
}
public class OdvozenaTrida : AbstraktniTrida
{
   public OdvozenaTrida(){}
}
```

Rozhraní, podobně jako abstraktní třídy, umožňují vytvořit dohodu o tom, které členy bude muset nově vytvářená třída definovat. K tomu, aby nově vytvářená třída byla nucena definovat členy rozhraní, musíme uvést, že třída konkrétní rozhraní implementuje. Na rozdíl od abstraktních tříd musí třída, která nějaké rozhraní implementuje, definovat všechny členy uvedené v rozhraní.

```
public interface INaseRozhrani
{
   void PrvniMetodaRozhrani();
   void DruhaMetodaRozhrani(int ciselnyParametr);
}

public class ImplementujiciTrida : INaseRozhrani
{
   public void PrvniMetodaRozhrani()
   {
      //Implementace metody
   }
   public void DruhaMetodaRozhrani(int ciselnyParametr)
   {
     //Implementace metody
   }
}
```

Relační operátory

Relační operátory využijeme v případech, kdy potřebujeme porovnat nějaké dvě hodnoty. Všechny operace, prováděné použitím relačních operátorů, mají výsledek logického typu **bool**

Tab. 1: Relační operátory

Operace	Výsledek
a == b	true, pokud se hodnota a rovná hodnotě b
a != b	true, pokud se hodnota a nerovná hodnotě b
a < b	true, pokud hodnota a je menší než hodnota b
a <= b	true, pokud hodnota a je menší než hodnota b, nebo je rovna hodnotě b
a > b	true, pokud hodnota a je větší než hodnota b
a >= b	true, pokud hodnota a je vetší než hodnota b, nebo je rovna hodnotě b

Logické operátory

Logické operátory se používají k provádění logických nebo bitových operací nad hodnotami.

Tab. 2: Logické operátory

Operátor	Popis
&	bitový součin obou operandů
|	bitový součet obou operandů
^	bitový výlučný součet obou operandů (XOR)
&&	logický součin dvou operandů
||	logický součet dvou operandů

Příkaz if

```
if (booleovský výraz)
{
   //příkazy, které jsou provedeny pouze, je-li podmínka splňena
}
else
{
   //příkazy, které jsou provedeny pouze, není-li podmínka splňena
}
```

Příkaz switch

Switch je příkaz pro mnohonásobné větvení programu. Každá větev musí být ukončena příkazem **break** nebo **goto**.

```
switch (výraz)
{
   case hodnota_1 :
     //příkazy pro hodnotu 1
   break;
   ...
   case hodnota_n :
     //příkazy pro hodnotu n
   break;
   default :
     //příkazy pro ostatni hodnoty
   break;
}
```

Prefixový a postfixový zápis inkrementace/dekrementace

++*operand* - hodnota je zvětšena o jedničku a je vrácena
operand++ - hodnota je nejdříve vrácena a potom je zvětšena o jedničku

Příklad:

```
int x = 5;
int y = 2;
x++; //x bude 6
y = ++x; // y bude 7, x bude 7
x = y++; // x bude 7, y bude 8
```

Příkazy pro vytváření cyklů

Jazyk C# nabízí k využití tří příkazů pro vytváření cyklů, kterými jsou **while**, **for** a **do-while**. S těmito příkazy jsou spojeny také příkazy **break** a **continue**, které slouží k ovlivnění průběhu cyklů.

break – tento příkaz ukončuje nejvnitřnější neuzavřenou smyčku cyklu a ihned opouští cyklus
continue – zapříčiní skok na konec nejvnitřnější neuzavřené smyčky, což znamená, že zbytek těla cyklu je vynechán, a pokračuje se další iterací cyklu

Příkaz while

Tento iterační příkaz testuje výraz s návratovou hodnotou bool vždy před průchodem cyklu. Pokud je výraz splněn, tedy jeho výsledek je **true**, pak je cyklus vykonán.Tím, že je testování logické podmínky prováděno na začátku, nemusí být cyklus vykonán ani jednou.

```
while (booleovský výraz)
{
   //tělo cyklu
}
```

Příkaz for

Definuje se počáteční hodnota, ukončující podmínku a způsob změny řídící proměnné cyklu. Příkaz for umožňuje, na rozdíl od cyklu while, tyto věci zadat přehledně na jedno místo. Typické použití příkazu for demonstruje následující příklad pro výpis čísel:

```
public static void VypisCisla()
{
   for (int i = 0; i < 10; i++)
      Console.WriteLine(i);
}
```

Struktury v C#

Struktury představují hodnotový datový typ. Použití struktur je vhodné při vytváření reprezentace jednoduchých objektů. Hlavní výhoda hodnotových typů je v jejich rychlé alokaci paměti a v nepřítomnosti režie, která je vlastní typům referenčním.

```
public struct Bod
{
   public int x,y;
   public Bod(int x, int y) //konstruktor s dvěma parametry
   {
      this.x = x;
      this.y = y;
   }
}
```

Konstruktory u struktur fungují odlišně od konstruktorů u tříd. V případě tříd je to tak, že dokud se nevytvoří nová instanci pomocí **new** a zavoláním příslušného konstruktoru, referenční proměnná má hodnotu **null**, tedy na žádnou instanci neukazuje a tím pádem není možné v danou chvíli objekt použít. U struktur neexistují odkazy, tak zavolání implicitního

(bezparametrického) konstruktoru pomocí **new** v jejich případě zajistí vytvoření nové instance, která má všechny své datové členy vynulovány. Není nutné použít **new** před použitím struktury, v tom případě ale je nutné, aby datové členy struktury byly inicializovány předtím, než jsou použity, jinak program nebude možné zkompilovat.

```
Bod b1;
b1 = new Bod(); //souřadnice x,y jsou nyní 0
```

Výčtové typy

Pro vytváření výčtových typů je v C# k dispozici klíčové slovo **enum**.

```
public enum DnyTydnu
{
   Pondeli,
   Utery,
   Streda,
   …
}
```

Deklarace proměnné typu dny v týdnu:

```
DnyTydnu den = DnyTydnu.Nedele;
```

Pole

Pole je datová struktura, která obsahuje určitý počet proměnných. Tyto proměnné jsou nazývány prvky pole. Prvky pole jsou indexovány a pomocí těchto indexů je později možné prvky z pole získávat. V jazyku C# jsou prvky pole indexovány od nuly. Všechny prvky pole jsou stejného typu. V prostředí .NET Frameworku je pole objektem referenčního typu, který je potomkem třídy **System.Array**.

Deklarace číselného pole typu int:

```
int[]ciselnePole=new int[4];
int[]ciselnePole=new int[]{1,2,3,4};//vložení prvků pole již při icializaci
int delkaPole = ciselnePole.Length;//je vrácena hodnota 4

Zjistění okrakajů pole:
int spodek = ciselnePole.GetLowerBound(0);
int vrsek = ciselnePole.GetUpperBound(0);
```

Cyklus foreach

Cyklus foreach je v jazyku C# předurčen k procházení polí a kolekcí. Cyklus provede definované operace pro každý prvek pole. Např. vypíše všechny prvky pole:

```
public static void VypisPole(int[] poleProVypsani)
{
  foreach(int prvek in poleProVypsani)
  {
   Console.WriteLine(prvek);//každý prvek bude vypsán
  }
}
```

Řetězce

Řetězce jsou v prostředí .NET Framework realizovány instancemi třídy **System.String**. Obsah jednotlivé instance této třídy je představován kolekcí, která obsahuje sekvenci objektů **System.Char**, tedy objektů jednotlivých znaků. Instance objektu typu **String** jsou neměnné. Změna řetězce probíhá tak, že je vytvořen nový pozměněný řetězec a původní je instance je zrušena.

```
string a = "Ahoj";
```

Tab. 3: Vybrané melody třídy string

Metoda	Popis
Clone()	vrátí referenci na daný řetězec
Compare()	porovná dva řetězce a vrátí počet stejných znaků
Contains()	pravda, jestli řetězec obsahuje zadaný řetězec
Copy()	vytvoří novou instanci řetězce jako kopii jiného řetězce
Insert()	vloží řetězec na danou pozici do jiného řetězce
Substring()	vytvoří podřetězec od zadané pozice

Třída StringBuilder

Je obsažena ve jmenném prostoru **System.Text**. Je vhodná pro provádění změn v objektech, které obsahují textový řetězec. Úpravy textových hodnot reprezentovaných instancemi třídy **StringBuilder** jsou mnohem méně náročnější na režii, než je tomu u hodnot reprezentovaných instancemi třídy System.String.

```
//vytvoří instanci pro max 50 znaků, inicializovanou řetězcem ABC
StringBuilder sb = new StringBuilder("ABC", 50);
//přidá tři znaky(D, E, F) na konec
sb.Append(new char[] { 'D', 'E', 'F' });
//vloží řetězec na začátek
sb.Insert(0,"Abeceda: ");
//nahradí malá k velkými K
sb.Replace('k', 'K');
//vypíše počet znaků v řetězci a samotný řetězec
Console.WriteLine("{0} chars: {1}", sb.Length, sb.ToString());
```

Tab. 4: Vybrané metody třídy StringBuilder

Metoda	Popis
Append()	přidá textovou reprezentaci objektu na konec konkrétní instance
Insert()	vloží textovou reprezentaci objektu na specifikovanou pozici v instanci
Remove()	od určité pozice vyjme určený počet znaků z instance
Replace()	nahradí specikovaný řetězec v instanci jiným specikovaným řetězcem

Výjimky v C#

V prostředí Microsoft.NET Frameworku je pro zpracovávání výjimečných - chybových stavů použit systém výjimek, který je představován třídou **System.Exception**. Výjimka je objekt alokovaný na tzv. hromadě, který nese informaci o chybovém stavu, který v aplikaci nastal a

pomocí zachycení tohoto objektu je možné nastalý stav nějakým způsobem zpracovat. Pokud v průběhu programu dojde k výjimečné situaci, vyhodí se výjimka. Vyhození výjimky znamená ukončení prováděného bloku programu a možnost výjimku ve volajícím bloku, kde již mohou být informace v dostačujícím množství pro napravení chyby, ošetřit.

Chráněné bloky a handlery výjimek

Chráněný blok, který se v jazyku C# uvozuje klíčovým slovem **try,** reprezentující blok ve kterém se zkoušejí provádět metody, které mohou vyhodit výjimku. Bezprostředně po tomto chráněném bloku může následovat jeden nebo více handlerů výjimky. Blok handleru je uvozen klíčovým slovem **catch.** U každého handleru je navíc uveden typ výjimky pro kterou je daný handler určen.

```
try
{
   //chráněny blok
}
catch (typVyjimky1 identifikator)
{
   //obslužný blok pro typVyjimky1
}
catch (typVyjimky2 identifikator)
{
   //obslužný blok pro typVyjimky2
}
catch (System.Exception ex)
{
   //společný obslužný blok pro všechny nespecifikované vyvolané výjimky
Console.WriteLine(ex.Message);//vypíše obsah výjimky
}
finally
{
   //závěrečné operace, které jsou provedeny vždy
}
```

V řadě případů je vhodné využít takzvané **závěrečné bloky - finally**. Na rozdíl od kódu uvnitř chráněného bloku **try**, jehož provádění se při vyvolání výjimky přeruší, tak závěrečné bloky jsou provedeny vždy, tedy ať k výjimce dojde nebo nikoliv.

Delegáty

Instance delegátů slouží k reprezentaci reference na metodu. Delegáty jsou někdy označovány jako „bezpečné ukazatele na funkce“, avšak na rozdíl od ukazatelů na funkce, známých z jazyku C++, jsou delegáty objektově orientované a typově bezpečné.

Deklarace delegáta pomocí klíčového slova **delegate**:

```
public delegate double MateOperDelegate(int a, int b);
public class MatOper //deklarace třídy a jejích metod
{
   public static double Soucet(int a, int b)
   {
      return a + b;
```

```
    }
    public static double Rozdil(int a, int b)
    {
       return a - b;
    }
}
```

Podmínkou je, aby delegát a metoda, na kterou delegát bude ukazovat, měli stejnou signaturu. V tomto případě je to `(int a, int b)`. Vytvoření instance delegáta:

```
MateOperDelegate novyDelegat = new MateOperDelegate(MatOper.Soucet);
```

Delegáty se často využívají jako parametry nějaké metody:

```
public class Kalkulator
{
   public void ProvedOperaci(int a, int b, MatOperDelegate operace)
   {
      Console.WriteLine(operace(a,b));} }
```

Konverze číselných datových typů v C#

V jazyku C# jsou konverze rozděleny na dva druhy a to na implicitní a explicitní. Implicitní konverze jsou takové konverze, při kterých není možné, aby došlo ke změně hodnoty, protože je hodnota konvertována na typ s větším rozsahem. Naproti tomu při explicitní konverzi je hodnota převáděna na typ s menším rozsahem hodnot a tím pádem může ke změně hodnoty dojít. Implicitní konverze jsou prováděny automaticky. Pro explicitní konverzi se v jazyku C# používá operátor **přetypování**, který je uveden v závorce, viz následující příklad:

```
int i;
long l;
i = (int)l;
s = (short)i;
b = (sbyte)s;
```

V určitých případech můžeme potřebovat kontrolovat, jesti při explicitním přetypování nedojde k přetečení hodnoty. Jazyk C# nabízí kontrolované převody použitím klíčového slova **checked**. Je-li hodnota určená pro explicitní přetypování větší než cílový datový typ, je vyhozena výjimka.

```
checked
{
   int intHodnota = 355;
   byte byteHodnota = (byte)intHodnota;
}
```

Konverze referenčních datových typů v C#

Třída **System.Object** je předkem všech tříd v .NET Frameworku a referenční proměnná tohoto typu může odkazovat na jakoukoli instanci.

```
Object objekt = new int[5];
int[] Pole = (int[])objekt;
Console.WriteLine(Pole.Length);
```

Explicitní konverzi je nutné použít všude tam, kde se snažíme referenční proměnnou obecnějšího typu převést na referenční proměnnou specifičtějšího typ. Tedy když chceme z reference na předka udělat referenci na potomka za účelem použití specifických členů, které na předkovi nejsou k dispozici.

I/O operace v C#

V prostředí .NET frameworku jsou všechny vstupně/výstupní operace realizovány pomocí takzvaných datových proudů, neboli streamů. Stream je abstrakce určité sekvence bajtů z nějakého zařízení, souboru, paměti nebo soketu TCP/IP. Takový datový proud je představován abstraktní třídou Stream ze jmenného prostoru **System.IO**.

1.3 Rozdělení datových typů

V .NET Frameworku se datové typy dělí na dvě skupiny. Jsou to:

Hodnotové typy (value types) – do této skupiny patří všechny číselné datové typy, typ char a ostatní struktury. U těchto jednoduchých typů se jejich hodnota ukládá přímo do proměnné – místa v paměti určené pro uložení hodnoty.

Referenční typy (reference types) – do této skupiny patří typ String a všechny třídy. Na rozdíl od hodnotových typů se jejich hodnota uloží do oblasti paměti nazývané halda. Do proměnné se uloží pouze adresa paměti, kde je hodnota uložena – reference.