Název závěrečné práce
Síťové protokoly a sockety

Bakalářská práce

Marek Balej

Vedoucí bakalářské práce: Ing. Ladislav Beránek, CSc., MBA

Jihočeská univerzita v Českých Budějovicích

Pedagogická fakulta

Katedra informatiky

2010

Rozsah grafických prací:

Rozsah pracovní zprávy: 60

Forma zpracování bakalářské práce: tištěná

Seznam odborné literatury:

1. Blum, R. C# Network Programming. London: Sybex, 2002.
2. Reid, F. Network Programming in .NET. London: Digital press, 2004.
3. Micriosoft MSDN [online]. 1995 [cit. 2009-04-02]. Dostupný z WWW: <http://msdn.microsoft.com>.
4. Poznáváme C# a Microsoft .NET [online]. 2001 [cit. 2009-04-02]. Dostupný z WWW: <http://www.zive.cz/Clanky/Poznavame-C-a-Microsoft-NET>.

Vedoucí bakalářské práce: Ing. Ladislav Beránek, CSc.
Katedra aplikované matematiky a informatiky

Datum zadání bakalářské práce: 7. dubna 2009

Termín odevzdání bakalářské práce: 30. dubna 2010

L.S.

doc. PhDr. Alena Hošpesová, Ph.D.
děkanka

PaedDr. Jiří Vaníček, Ph.D.
vedoucí katedry

V Českých Budějovicích dne 7. dubna 2009

Rozsah grafických prací:
Rozsah pracovní zprávy: 60
Forma zpracování bakalářské práce: tištěná

Seznam odborné literatury:

1. Blum, R. C# Network Programming. London: Sybex, 2002.
2. Reid, F. Network Programming in .NET. London: Digital press, 2004.
3. Micriosoft MSDN [online]. 1995 [cit. 2009-04-02]. Dostupný z WWW: <http://msdn.microsoft.com>.
4. Poznáváme C# a Microsoft .NET [online]. 2001 [cit. 2009-04-02]. Dostupný z WWW: <http://www.zive.cz/Clanky/Poznavame-C-a-Microsoft-NET>.

Vedoucí bakalářské práce: Ing. Ladislav Beránek, CSc.
Katedra aplikované matematiky a informatiky

Datum zadání bakalářské práce: 7. dubna 2009
Termín odevzdání bakalářské práce: 30. dubna 2010

doc. PhDr. Alena Hošpesová, Ph.D.
děkanka

L.S.

PaedDr. Jiří Vaníček, Ph.D.
vedoucí katedry

V Českých Budějovicích dne 7. dubna 2009

Prohlášení

Prohlašuji, že svoji bakalářskou práci jsem vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.

Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své bakalářské práce, a to v nezkrácené podobě elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách.

V Českých Budějovicích dne

Anotace

Práce se bude zabývat síťovými protokoly a sockety a jejich využitím v programovacím jazyce C#. Bude se tedy zabývat programováním síťových aplikací na platformě .NET od firmy Microsoft a nástroji, které nám k tomu jazyk C# poskytuje. Budou popsány nástroje a metody programování síťových aplikací a ukáže vytvoření a popis ukázkových aplikací pracujících se sockety a aplikačními protokoly.

Abstract

My work will deal with network protocols and sockets and their use in programming language C#. It will therefore deal programming network applications on the platform .NET from Microsoft and instruments, which C# provides to us. There will describe the tools and methods for programming network applications, and shows a description and sample applications that work with sockets and application protocols.

Poděkování

Rád bych poděkoval vedoucímu mé bakalářské práce Ing. Ladislavu Beránkovi za cenné rady a připomínky.

Obsah

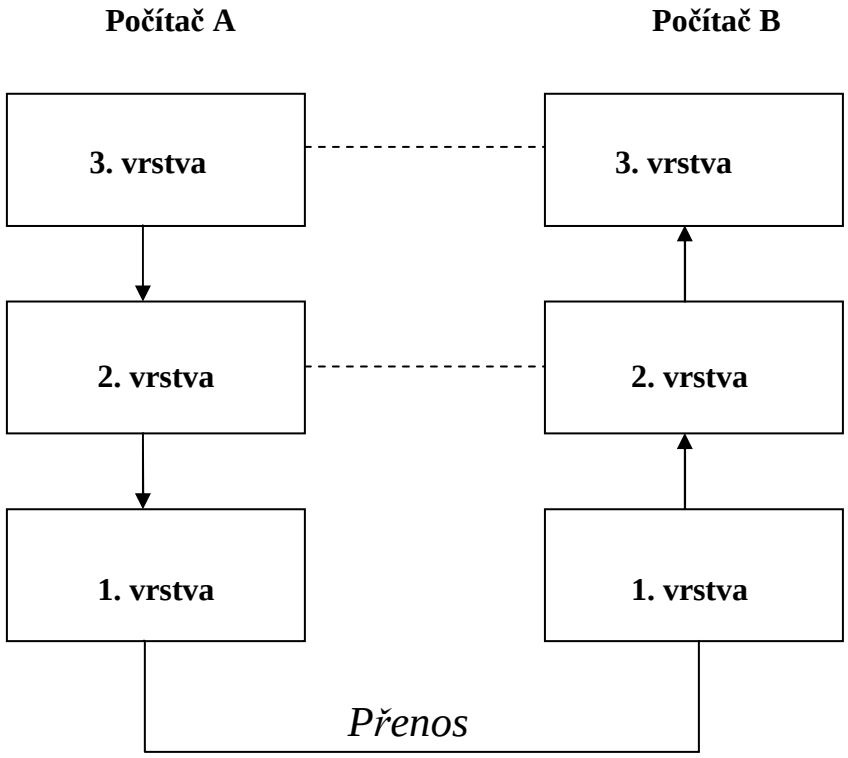
1 Počítačové sítě

1.1 Síťový model

Síťový model je v podstatě popis struktury komunikace mezi počítači. To znamená, že jednoznačně popisuje, jak mají být data upravena pro přenos a následně jak mají být na druhé straně přečtena. Je třeba, aby tato komunikace byla standardizována, zejména kvůli spolupráci síťových prvků (routerů, switchů….), ale i počítačů od různých výrobců, v opačném případě by nebylo možné v síti jednotlivé produkty kombinovat, což by zejména v rozsáhlejších sítích představovalo velký problém.

Síťový model rozděluje komunikaci do několika vrstev. Počet vrstev je v každém modelu jiný, ale o tom více až později.

Nejprve si řekneme, proč síťové modely komunikaci takto dělí. Problematika síťové komunikace je velice široká, obsahuje rozmanitou škálu služeb. Je tedy vhodné přidělit každé vrstvě určitý druh funkcí, který bude zahrnovat, a ostatní přenechá dalším vrstvám [3]. To znamená, že na každé straně komunikačního kanálu spolu komunikují pouze stejné vrstvy. Samozřejmě, že ve skutečnosti spolu stejné vrstvy nekomunikují přímo („horizontálně“), ale komunikují přes ostatní vrstvy, tedy vertikálně (viz. následující obrázek).

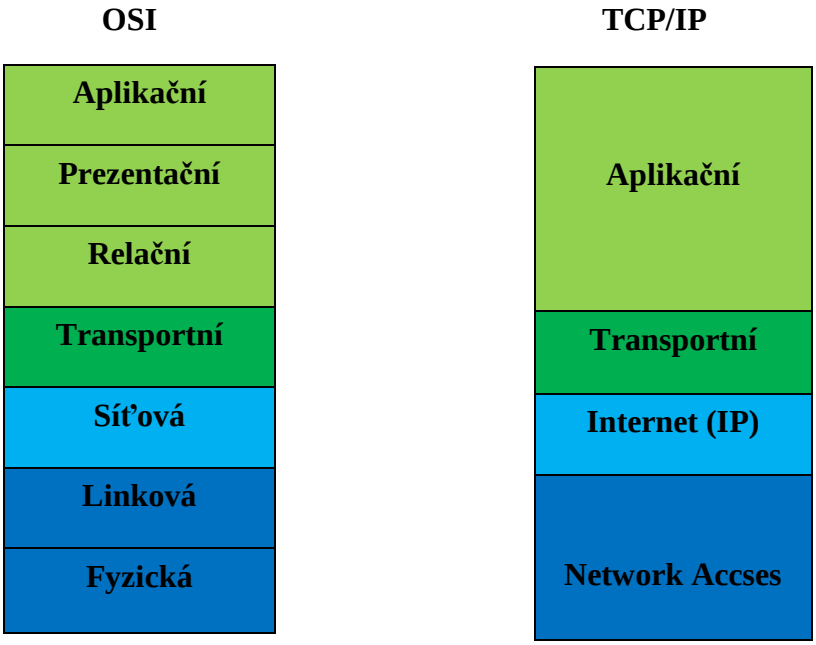


Obr. 1: Vrstvy síťové komunikace

Řekněme tedy, že počítač A posílá paket počítači B, data zpracuje nejprve každá vrstva počítače A, sestupně od nejvyšší. Následně je zpráva fyzicky přenesena, tento přenos má na starosti nejnižší vrstva modelu. Na druhé straně (počítač B) se data zpracovávají v opačném pořadí od nejnižší vrstvy. Každá vrstva ale zpracuje pouze tu část, která je určená pro ni, tedy pouze ty údaje, které stejná vrstva na druhé straně ke zprávě přidala.

Na síťové modely lze také pohlížet jako na soustavu protokolů. Protokol je v informatice standard, který definuje průběh komunikace a přenosu dat mezi dvěma koncovými body [6]. Protokol je v podstatě služba, kterou nám poskytuje daná soustava (síťový model) a která pracuje na určité vrstvě modelu. To, na jaké vrstvě pracuje, je dáno jeho funkcí.

V současné době jsou nejznámější dva modely. Prvním z nich je OSI (Open System Interconnection) model navrhnutý organizací ISO (International Stanadart Organization) a druhým TCP/IP. TCP/IP má čtyři vrstvy a složitější OSI model má vrstev sedm.



Obr. 2: struktura síťových modelů OSI a TCP/IP

1.1.1 ISO OSI

OSI je model, navržený pro komunikaci otevřených systémů. Rozděluje komunikaci na síti do sedmi vrstev, přičemž se tyto vrstvy dají rozdělit do dvou velkých bloků. Blok tří nejvyšších vrstev se zaměřuje na aplikace a jejich potřeby a blok tří spodních vrstev je zaměřen na přenos dat na síti. Mezi těmito bloky je ještě transportní vrstva. Vrstvy jsou na sobě navzájem nezávislé [7].

Dále si popíšeme jednotlivé vrstvy OSI modelu.

Fyzická vrstva

Do fyzické vrstvy patří například kabely, konektory, definice kmitočtů atd. Má na starosti fyzický přenos dat mezi dvěma koncovými body. Jakým způsobem to bude provedeno, záleží na použité technologii (optické kabely, metalické kabely, bezdrátový přenos). Dále má za úkol převést jednotlivé bity do podoby potřebné pro daný druh přenosu (elektrické impulsy, radiové vlny…) a na druhé straně informaci zpětně interpretovat pro vyšší vrstvy.

Linková vrstva

Linková vrstva má na starost přenos paketů vyšších vrstev. Základní jednotkou této vrstvy je datový rámec. Ten se skládá ze záhlaví, přenášených dat a zápatí. Přenášená data jsou zpravidla pakety vyšších vrstev, k nim vrstva přidá fyzickou adresu cílového zařízení, synchronizační sekvenci pro síťové adaptéry (záhlaví) a v zápatí nese kontrolní součet pro zjištění správnosti přenesených dat [5]. Tyto funkce má na starost firmware síťového adaptéru, jehož součástí je také již výše zmíněná fyzická adresa (známá též jako MAC adresa) adaptérů, která je nejvýznamnější součástí datového rámce. Na základě MAC adres filtrují provoz na síti switche.

Síťová vrstva

Linková vrstva umí své rámce přenášet pouze za předpokladu, že je mezi dvěma koncovými body přímé spojení. Pokud je mezi počátečním a cílovým počítačem jeden nebo více uzlů, je zapotřebí vybrat správnou cestu. Výběr této cesty má na starosti síťová vrstva. K tomu využívá logických adres, které jsou obsaženy v záhlaví síťového paketu. Uzly, přes které pakety putují, jsou realizovány routery (směrovači), pro které linková vrstva vybalí síťový paket, síťová vrstva následně rozhodne, kudy poslat paket dál. Tento paket pak opět

předá linkové vrstvě atd. Toto se opakuje do doby, než data dorazí ke svému cíli [15].

Transportní vrstva

Transportní vrstva má na starosti přípravu dat pro bezchybný přenos. Délka zprávy se může lišit, proto ji rozdělí na pakety (ty se stejně jako u síťové vrstvy skládají ze záhlaví a zápatí). Pakety se očíslují a také se k celé zprávě vypočítá kontrolní součet. Na přijímací straně lze pak zjistit, zda některé pakety nechybí, popřípadě si vyžádat jejich opětovné vyslání.

Transportní vrstva se nezajímá o spojení mezi dvěma počítači (v tom se spoléhá na nižší vrstvy), ale navazuje spojení mezi dvěma aplikacemi na vzdálených počítačích [3]. Aplikace jsou v rámci jednoho počítače jednoznačně definovány, a to tzv. porty.

Relační vrstva

Tato vrstva má za úkol navazování spojení (relací) mezi zdrojovým a cílovým bodem. Určuje se zde, jakým způsobem bude spojení probíhat (full duplex, half duplex), dále např. jak bude řešeno násilné ukončení atd. [5]. Rozdíl mezi touto a transportní vrstvou je také v tom, že v případě čtvrté vrstvy se spojení navazuje pouze na dobu, kdy jsou přenášena data. V případě relací je spojení navázáno i v případě, že žádná komunikace neprobíhá. Příkladem může být síťový disk, relace s ním trvá po celou dobu spojení. Nižších vrstev se využívá, pouze pokud na disk začneme nahrávat nebo z něj stahovat data.

Prezentační vrstva

Prezentační vrstva má na starost přípravu dat pro přenos, jejich zabezpečení, popřípadě kompresi a to musí provést tak, aby byla na druhé straně data

správně přečtena. Obecně se jedná o to, aby pro příjemce znamenala data to samé co pro odesílatele. Dva počítače mohou například využívat jiné kódování (ASCII, EBCDIC) a musí být tedy provedena určitá konverze. A právě tato konverze je hlavním (ale ne jediným) úkolem prezentační vrstvy [24].

Aplikační vrstva

Aplikační vrstva tvoří jakousi bránu nebo rozhraní pro aplikace chtějící využívat referenčního modelu OSI. Původně měla zahrnovat všechny a celé aplikace, které chtějí se sítí spolupracovat. Postupem času se to ukázalo jako nesmyslné a prakticky nemožné. Byly tedy standardizovány pouze části aplikací, u kterých je to nezbytně nutné, a zbytek aplikací byl vysunut nad aplikační vrstvu (např. GUI) [9]. Mezi typické služby patří přenos souborů a elektronická pošta. Například poštovní klient (jeho vzhled, ovládání) je již mimo OSI model, naopak to v jakém formátu zprávu musí poslat, aby ji ostatní klienti přijali, je dáno aplikační vrstvou, ale to, jak se zprávou nakládá, je individuální záležitostí každého klienta.

1.1.2 TCP/IP

Pod pojmem TCP/IP si většina lidí představí právě protokoly TCP a IP. Tento pojem ale představuje mnohem více a zahrnuje celou řadu protokolů, i když tyto dva protokoly jsou jeho základem. Tato síťová struktura je v současné době základem internetu.

Rodina protokolů TCP/IP byla vyvinuta pro síť ARPANET, která měla sloužit ministerstvu obrany v USA. Původním protokolem pro komunikaci v této síti byl NCP (Network Control Protocol), ten však byl postupem času nahrazován propracovanějším modelem TCP/IP. TCP/IP model vznikl jako výsledek projektu prováděného agenturou DARPA, který zkoumal technologie

propojování různých typů sítí. Vyústěním byla „síť sítí“, která vyšla pod označením Internet. Postupem času se k Internetu připojovalo stále více sítí, až se rozšířil po celém světě [10].

Network Access

V TCP/IP se první vrstva nazývá Network Access, někdy také Network Interface layer (vrstva síťového rozhraní). Obecně architektura TCP/IP tuto vrstvu příliš neřeší. Je v podstatě jedno, jaké technologie je na této vrstvě použito. V případě dvoubodového spoje ji může tvořit obyčejný ovladač. V jiných případech to může být velmi složitý systém s vlastním linkovým protokolem. Velmi často se zde využívají protokoly OSI modelu, který má linkovou vrstvu specifikovanou. Technologií, která se zde často využívá, je např. Ethernet [15].

Internetová vrstva

Bezprostředně vyšší vrstvou vrstvy síťového rozhraní je vrstva internetová. Její úkol je stejný jako úkol vrstvy síťové u OSI modelu, tzn. dostat pakety tam, kam se dostat mají. Této vrstvě se také někdy říká IP vrstva, protože je realizována protokolem IP (Internet Protokol). K tomu využívá tzv. IP datagramu, který ve svém záhlaví nese IP adresu. Každé síťové rozhraní má v internetu jedinečnou IP adresu, je tedy jednoznačně dáno, kam má být datagram doručen. O IP protokolu, jeho adresách a možnostech si povíme více později.

Transportní vrstva

Stejně jako vrstva internetová i tato vrstva je v podstatě ekvivalentní vrstvě z OSI modelu, v tomto případě vrstvě transportní. Této vrstvě se někdy také

říká TCP vrstva. Podle mého názoru je tento název trochu zavádějící, protože transportní vrstvu nemusí tvořit pouze protokol TCP, ale může zde běžet i protokol UDP. Tato vrstva zajišťuje komunikaci mezi koncovým a cílovým bodem, kterým v tomto případě nejsou cílové počítače, ale přímo aplikační procesy, které jsou v rámci daného počítače identifikovány pomocí portů. Podle použitého protokolu může být komunikace realizována spojově (TCP) nebo nespojově (UDP). Vzhledem k tomu, že tyto protokoly mají také na starost kontrolu správnosti přenesených dat, je v případě nesprávného přijetí (chybějící paket, zkomolený paket) u spojové komunikace vyžádáno opětovné vyslání, u nespojové služby se chyby ignorují a chybné pakety zahazují. Více si o těchto protokolech, zejména o TCP povíme také v následující kapitole.

Aplikační vrstva

Aplikační vrstva odpovídá hned několika vrstvám ISO/OSI. Konkrétně se jedná o vrstvu prezentační, relační a aplikační. Je tedy jasné, že v případě prezentačních a relačních služeb si musí aplikace pomoct samy [8]. Aplikačních protokolů je celá řada a některými z nich se budu zabývat i v praktické části.

1.2 Rodina protokolů TCP/IP

Protokol je norma (standard), která popisuje, jakým způsobem bude probíhat komunikace mezi počítači na síti. V internetu se používají normy RFC (request for comments). Všechny současné normy lze najít na stránkách http://www.ietf.org/rfc.html. Jednou přidaná norma se již neodstraňuje, proto jich je zde hodně zastaralých. Výhodou je, že normu může přidat prakticky každý, v čemž je rozdíl např. od norem vydaných organizací ISO.

Jak jsem se již výše zmiňoval, na jednotlivé síťové modely lze pohlížet také jako na rodinu (soustavu) protokolů, které v podstatě představují služby, jimiž daný síťový model disponuje. Podle funkce, ke které protokoly slouží, jsou rozděleny do jednotlivých vrstev. Jak již víme, u TCP/IP jsou vrstvy čtyři. Vzhledem k tomu, že tento model příliš neřeší nejnižší vrstvu Network Access a ani vzhledem k zaměření práce pro nás není příliš zajímavá, budu se zabývat až protokoly vyšších vrstev.

1.2.1 Protokoly Internetové vrstvy

1.2.1.1 IP protokol

IP protokol je úplným základem TCP/IP modelu. Slouží ke směrování dat, a to i přes více uzlů, tzn., že je schopen data směrovat i do jiných sítí.

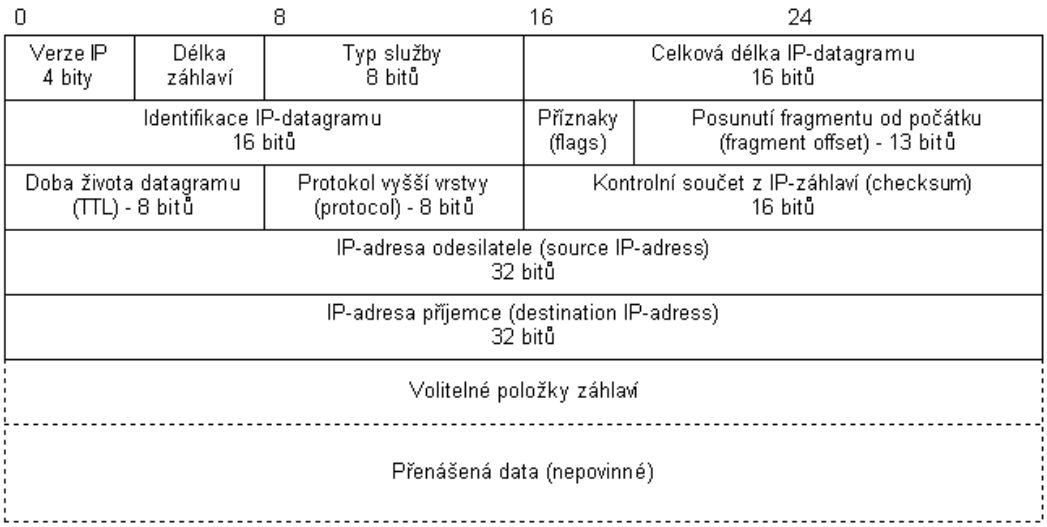
Právě po IP protokolu dostal Internet své jméno (InterNet Protokol). Tento protokol je tvořen dílčími protokoly (samotný IP a dále, IGMP, ARP a RARP). U ARP a RARP není toto tvrzení tak přesné. Vzhledem k tomu, že tyto dva protokoly jsou samostatnou službou a jejich pakety jsou baleny rovnou do rámce linkového protokolu, jsou to spíše pouze protokoly pracující ve stejné

vrstvě [3]. Nicméně je pravdou, že tyto protokoly úzce spolupracují. Stručně si tyto protokoly představíme později.

1.2.1.1.1 IP-datagram

Již výše v první kapitole o síťových modelech jsem zmiňoval, že základní jednotkou IP je IP-datagram. Pojďme se na něj podívat trochu podrobněji.

Na obrázku je nákres struktury IP-datagramu.



Obr. 3: IP datagram [14]

Nyní si jednotlivé části představíme.

Verze – verze IP protokolu – 4 nebo 6.

Délka záhlaví – délka záhlaví diagramu. Toto číslo je uváděno v násobcích 4 Byte.

Typ služby – tato část byla původně myšlena jako možnost přidělování různých priorit jednotlivým paketům, to se v praxi ale příliš neujalo.

Celková délka IP datagramu – udáváno v bajtech. Maximální délku v podstatě určuje délka tohoto pole 16 bitů – 65535.

Identifikace – vkládá ji operační systém odesilatele. Využívá se k fragmentaci diagramů.

TTL – Time To Live. Doba života datagramu. Zamezuje nekonečnému toulání paketu po síti. Je určena počtem skoků. Na každém směrovači se hodnota o jedničku sníží. Při nulové hodnotě se paket zahazuje.

Protokol vyšší vrstvy- číselné označení protokolu vyšší vrstvy, který využívá IP protokolu pro transport svých paketů. Obvykle jím bývá TCP, popřípadě UDP, ale mohou jím být i protokoly ICMP a IGMP, které jsou součástí IP protokolu.

Kontrolní součet – počítá se pouze ze záhlaví. Při změně záhlaví (např. TTL) jej musí směrovač přepočítat.

IP-adresa odesílatele (příjemce) – obsahuje IP adresu odesilatele (příjemce) [3].

1.2.1.1.2 IP adresa a subnety

Počítače komunikující pomocí TCP/IP modelu jsou v síti jednoznačně identifikovány pomocí IP adresy. Tato adresa je 32 bitová a pro snazší zapamatovatelnost je také uváděna čtyřmi dekadickými číslicemi, které se oddělují tečkou. Tato čísla logicky nabývají hodnot odpovídajících 8 bitům, tedy 0-255 (př. 192.168.0.55). Část této adresy je vyhrazena na pro identifikace sítě (tzv. Net ID), druhá část slouží k identifikaci počítače v dané síti (tzv. Host ID).

K tomu, abychom mohli rozdělit adresu na dvě výše uvedené části, se využívá síťová maska. Jedničky v masce znamenají, že bity na stejné pozici v adrese reprezentují Net ID, a nuly zase Host ID. Tímto způsobem je možné přidělovat bloky adres podle toho, jak velká je síť. U třídy A může být na jedné síti obrovské množství počítačů, na druhou stranu je takovýchto sítí značně

omezené množství. V případě třídy C zase máme k dispozici velké množství těchto sítí, ale v každé síti máme k dispozici pouze 255 adres. Původně existovaly tři třídy (viz obrázek).

0	8	16	24
Třída A	Net ID	Host ID	
Třída B	Net ID		Host ID
Třída C	Net ID		Host ID

Obr. 4: Základní rozdělení sítí do tříd [5]

O jaký typ sítě se jedná, poznáme vždy z počátečních bitů adresy (třída A má první bit 0, u třídy B mají první dva bity hodnotu 10 a všechny adresy třídy C začínají kombinací 110).

Takovéto rozdělení do tříd ovšem neodpovídá dnešním potřebám. Zejména u poskytovatelů internetového připojení je třeba si uvědomit, že počet adres, které mají k dispozici, je omezený. Bylo by tedy velmi nehospodárné přidělit každému uživateli celou síť třídy C, stejně tak je nežádoucí připojit více uživatelů do stejné sítě, proto se dělí sítě na tzv. subsítě (podsítě).

Rozdělení se provádí pomocí změny masky. Jednoduše přesuneme bity značící Host ID do části Net ID a to tak, že jejich hodnoty v masce změníme z nuly na jedničky. Tyto jedničky “navíc“ označují bity, které budou určovat adresu podsítě. Takovýto přesun lze provádět i opačným směrem, tedy z jedničky na nulu. V tomto případě dochází ke spojení více menších sítí a vzniká tzv. supersíť [5].

1.2.1.2 Protokol ICMP (Internet Control Message Protocol)

Protokol ICMP je součástí protokolu IP jeho pakety jsou tedy baleny do IP paketu. Slouží k signalizaci mimořádných událostí. Těchto událostí je celá řada, ovšem skutečnost je taková, že konkrétní implementace TCP/IP podporují vždy jen jistou část těchto signalizací [3]**.** Jednotlivé druhy signalizací mají každá své číselné označení a podle tohoto čísla poznáme, o jaký typ signalizace se jedná, např. nedosažitelný uzel, neznámá adresa, nedosažitelný port atd.

Velmi často využívanou službou ICMP je “žádost o echo”. Ta slouží zejména k zjišťování konektivity neboli průchodnosti sítě. ICMP paket s žádostí o echo se vysílá pomocí příkazu **ping**, který mají implementovaný všechny operační systémy pracující s TCP/IP. Cílový uzel pak odpovídá paketem typu “echo”.

Hlavička ICMP obsahuje vypadá takto:

Typ(8 bitů)	Kód(8 bitů)	Kontrolní součet(16 bitů)
Data		

Obr. 5: ICMP paket [11]

Dalším protokolem, který je součástí IP protokolu, je IGMP (Internet Group Managment Protocol). Tento protokol využívají zejména směrovače. S jeho pomocí si získávají informace o skupinách pro vícesměrové vysílání (*multicasting*). Obsahem IGMP paketu může být např. žádost o zařazení do skupiny, opuštění skupiny atd. Data jsou poté zasílána všem členům skupiny.

1.2.2 Protokoly Transportní vrstvy

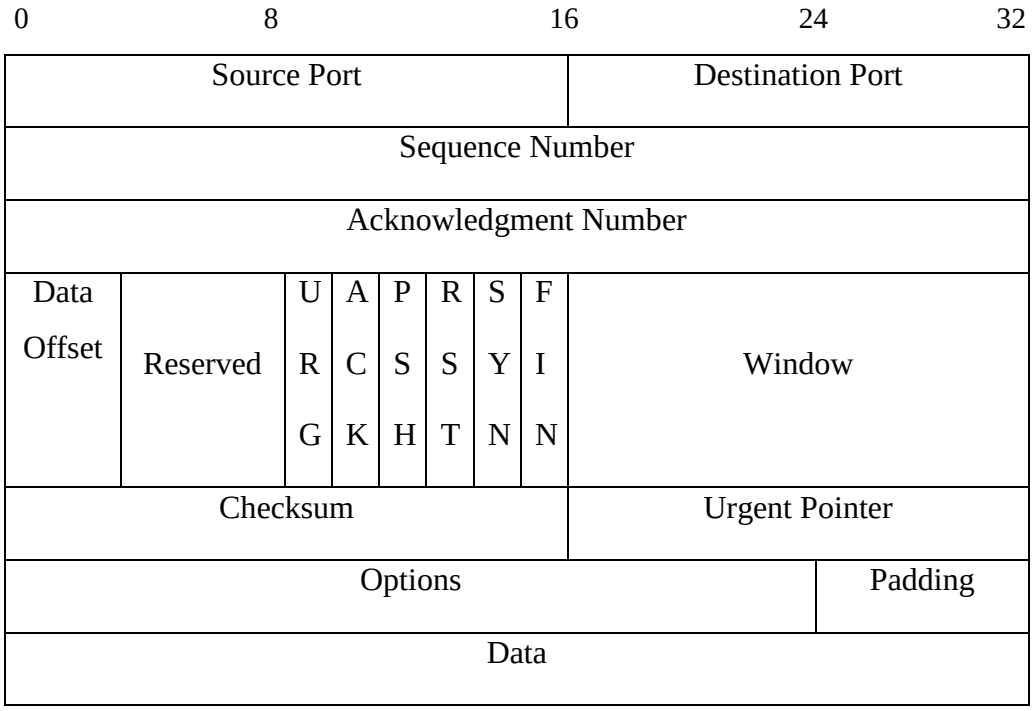
1.2.2.1 TCP

Protokol TCP (Transmission Control Protocol) je protokolem, který přenáší data mezi aplikacemi. Na rozdíl od protokolu IP, který adresuje síťové rozhraní, TCP adresuje jednotlivé aplikace, a to za pomoci tzv. portů. TCP je spojovanou službou. Po navázání spojení mezi aplikacemi se vytvoří virtuální okruh. Právě díky tomuto spojení může aplikace pomocí TCP vyžádat opětovné vyslání ztraceného, popřípadě zkomoleného paketu. Jestli nějaký paket chybí, nebo je chybný, TCP to zjistí pomocí očíslování paketů a kontrolního součtu (tyto položky jsou součástí záhlaví TCP paketu).

Konce spojení jsou jasně definovány porty. Port není v podstatě nic jiného než číslo, které nám umožňuje odlišit od sebe aplikace využívající síťové rozhraní. Podle těchto čísel zjistíme, ke kterému procesu data patří. K dispozici máme čísla 0 až 65535 (2 bajty) pro protokol TCP (protokol UDP má svoji sadu čísel také 0 až 65535) [16].

TCP segment

Základní datovou jednotkou je paket, někdy také nazývaný segment, který se také skládá ze záhlaví a datové části.



Obr. 6: TCP paket [12]

Source port – zdrojový port.

Destination port – cílový port.

Sequence number – pořadové číslo prvního bajtu v odeslaném segmentu.

Acknowledgment number – pořadové číslo prvního bajtu následujícího očekávaného segmentu.

Data offset – délka záhlaví.

Reserved – rezervované šestibitové pole pro budoucí použití. Musí být nulové.

Control bits – pole příznaků. Určují specifické vlastnosti segmentu.

Window size – velikost okna. Počet oktetů potvrzovaných najednou.

Checksum – kontrolní součet (nepovinný).

Urgent pointer – platí v případě, že je nastaven příznak URG. Ukazuje na poslední oktet “urgentních dat”.

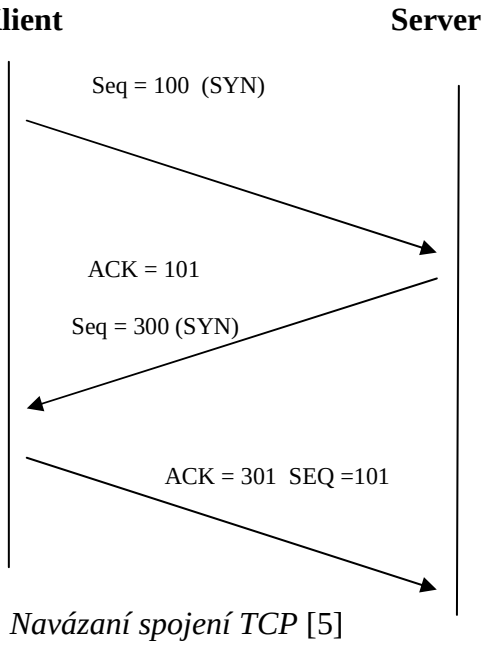
Options – pole volitelných parametrů TCP. Např. maximální délka segmentu.

Padding – "zarovnání segmentu". Specifické pole obsahující tolik nulových bitů, aby délka záhlaví segmentu byla násobkem čísla 32.

Přenos dat

TCP se snaží, aby bezprostředně vyšší vrstvě nabízel přenos jednotlivých bajtů (v tomto případě velmi často nazývaných oktety). TCP tedy na straně odesílatele předpokládá, že aplikační vrstva mu bude zprávu předávat po jednotlivých oktetech a na straně příjemce si je bude zase v této podobě vyzvedávat. Ovšem toto je spíše iluze vytvořená pro vyšší vrstvu právě protokolem TCP. Samotný přenos TCP nerealizuje po jednotlivých oktetech, ale střádá je na straně odesilatele ve speciální paměti (bufferu). Poté co objem dat dosáhne kapacity jednoho segmentu, přidá k nim TCP své záhlaví (vznikne TCP segment) a předá data nižší vrstvě. Existují ale i případy kdy takovýto způsob není žádoucí, protože některé aplikace vyžadují okamžitou odezvu. Například při práci na vzdáleném počítači pomocí protokolu telnet, je výstup z klávesnice směrován přes síť na cílový počítač a ten následně výstup směruje na obrazovku počítače, pomocí kterého se vzdálený počítač připojuje. Kdybychom v tomto případě čekali na zaplnění bufferu, uživatel by musel nejdříve zadat několik příkazů, než by TCP odeslal svůj segment a celou tuto dobu by se uživatel nedočkal jakékoliv odezvy. Pro tyto situace existuje mechanismus **push**, kterým si může aplikace vynutit odeslání dat, aniž by byl buffer plný [8].

Navázání spojení



Obr. 7: Navázaní spojení TCP [5]

Na obrázku je znázorněno navázání komunikace mezi klientem a serverem. Protokol TCP nemusí vždy zajišťovat služby pouze aplikacím typu klient-server, ale tento případ je běžnější. Aplikace na serveru tedy naslouchá na svém portu. Pokud bude chtít klient se serverem komunikovat, vygeneruje si počáteční pořadové číslo odesílaného bajtu (sequnce number), v našem případě 100. Tento jediný segment nemá nastavený příznak ACK, protože nepotvrzuje žádné přijaté oktety a má nastaven příznak SYN (synchronizační segment – jde o první segment, který klient vyslal). Server tento segment přijme a vygeneruje vlastní počáteční pořadové číslo, které také uvede v záhlaví segmentu. Tento segment bude již obsahovat ACK (Acknowledgement number), kterým v podstatě potvrzuje správné přijetí dat od klienta. Toto ACK je o jedničku vyšší než SEQ přijatého datagramu (dalo by se říci, že toto číslo znamená, s jakým SEQ čeká strana příjem následujícího segmentu). I tento segment bude mít nastaven příznak SYN. V záhlaví těchto dvou prvních segmentů se často nachází volitelná položka MSS (Maximum Segment Size), čímž jedna strana

druhé říká, jakou maximální velikost TCP segmentu by měl její protějšek odesílat, aby nemuselo docházet k fragmentaci IP datagramu.

Následně ještě klient potvrdí přijetí potvrzení. Tedy SEQ se stejnou hodnotou, jako bylo ACK přijatého, a s ACK o jedničku vyšším, než bylo SEQ přijatého segmentu. Tímto segmentem končí navazování spojení. Ani jeden z těchto segmentů v sobě nenese žádná aplikační data. U dalších segmentů tomu již tak většinou bude, i když být nemusí. Může se totiž stát, že jedna strana nemá žádná data k odeslání, a tak pouze vyšle žádost o další data. Segmenty, které nesou aplikační data, obsahují příznak PSH. Vzhledem k tomu, že TCP potřebuje k navázání spojení odeslat dohromady tři segmenty, jedná se o tzv. *trojfázový handshaking*.

Z výše napsaného tedy vyplývá, že jedna strana podle ACK v přijatém segmentu snadno zjistí správný příjem dat na druhé straně. Ovšem při chybném obdržení (popřípadě neobdržení) segmentu příjemce nijak nereaguje a chybné segmenty zkrátka zahodí, tzn., že odesilatel nemá tušení o chybném příjmu na druhé straně. Z tohoto důvodu protokol TCP nastavuje časový limit určující, dokdy musí být přijato potvrzení. Pokud v tomto limitu potvrzení nedorazí, jsou data odeslána znovu [3].

Ukončení spojení

Ukončit spojení mohou obě dvě strany (klient i server). Ukončení probíhá pomocí segmentů s příznakem FIN. Na rozdíl od zřizování spojení je jeho ukončení čtyřfázové (je třeba odeslat čtyři segmenty).

1. První strana odesílá segment s příznakem FIN. Tím dává najevo, že chce ukončit spojení. Jde o tzv. aktivní ukončení spojení. Strana, která toto ukončení provede, již nemůže odesílat žádná data.
2. Druhé straně nezbývá nic jiného než toto ukončení potvrdit (pasivní ukončení spojení – potvrzení segmentem bez příznaku FIN). Avšak může stále

pokračovat v odesílání dat, dokud sama spojení neukončí. Tomuto stavu říká polozavřené spojení.

3. Druhá strana již také nemá data k odeslání, a tak tedy signalizuje úplné ukončení spojení (segment s příznakem FIN).

4. První strana tento segment potvrzuje a obě strany končí s komunikací [3].

Pro úplnost bych ještě doplnil, že spojení je také možno ukončit ihned, a to pomocí příznaku RST (Connection Reset). Pokud druhá strana obdrží segment s tímto příznakem, musí okamžitě ukončit spojení.

Důvodů k resetu může být více. Příkladem může být adresování segmentů na prázdný port (port, na kterém neběží server). Dalším případem může být odmítnutí spojení ze strany serveru. Pokud se mu zdá klient, který se chce připojit nebo už je připojen, nedůvěryhodný (nedostatečné šifrování, nedokáže se autentizovat), okamžitě ukončuje spojení.

Pro zajímavost se ještě zmíním o tzv. **resetovacích útocích**. Resetovací útok je útok provedený pomocí segmentu s nastaveným příznakem RST.

Útokem můžeme oběti ukončit spojení se serverem, a to zasláním RST segmentu na port, na kterém běží klient [17].

Zde se trochu projevují bezpečnostní nedostatky TCP protokolu. Útoků vedených pomocí TCP existuje více, např. SYN flooding nebo hijacking.

1.2.2.2 UDP

Protokol UDP je druhou alternativou k protokolu TCP. Je to jednodušší služba, která je nespojovaná. To znamená, že mezi dvěma koncovými body nenavazuje spojení. Nijak tedy nepotvrzuje přijaté pakety a není tedy možné zjistit, zda se nějaký paket neztratil. V tomto případě se o nápravu musí postarat aplikační vrstva.

UDP datagram

0 16 32

Source port	Destination port
UDP lenght	Checksum
Data	

Obr. 8: UDP diagram [13]

Jak je vidět na obrázku, datagram protokolu UDP je opravdu velmi jednoduchý. Obsahuje pouze zdrojový a cílový port komunikace. Informaci o délce datagramu, kontrolní součet a samozřejmě samotná data. Navíc kontrolní součet u UDP je nepovinný.

U čísel portů je třeba zmínit, že UDP má vlastní sadu tzn., že například port 32 UDP je odlišný od portu 32 TCP.

Využití UDP

Jak jsem již zmínil, UDP je službou nespolehlivou, což je např. při přenosu souborů nežádoucí. Pokud by nám část souboru nedorazila, byly by všechny ostatní části bezcenné. Ovšem existují aplikace, kde si tato služba své uplatnění najde. Velkou výhodu je, že u UDP nemusíme adresovat pouze konkrétní síťové rozhraní, ale můžeme adresovat více stanic najednou pomocí tzv. multicast (adresné oběžníky). Kdybychom chtěli něco podobného realizovat pomocí TCP, musela by každá stanice navázat spojení se serverem.

UDP tedy využijeme především u takových aplikací, u kterých potřebujeme co nejmenší režii a u kterých není tak nutné, aby všechna data dorazila v pořádku. Takovým příkladem může být streaming (velmi často uváděným příkladem jsou internetová radia).

1.2.3 Aplikační protokoly

Jak jsem se již zmiňoval, v TCP/IP modelu neexistuje relační ani prezentační vrstva. Zejména absenci prezentační vrstvy bylo třeba nějakým způsobem řešit. Šifrování se tedy provádí pomocí speciálních prezentačních-aplikačních protokolů [3]. Příkladem může být SSL (Secure Socket Layer). Nejedná se až tak přímo o službu aplikační vrstvy, ale je to spíše vrstva vložená mezi aplikační vrstvu a vrstvu transportní. Pro kódování se např. využívá protokol SSL.

Čistě aplikační protokoly tedy využívají samotné aplikace [3].

1. Uživatelské protokoly – těmi se rozumí protokoly, se kterými pracují uživatelské aplikace. Příkladem mohou být FTP, SMTP, HTTP….

2. Služební protokoly – protokoly, se kterými se uživatel nesetká. Jsou velmi často využívané směrovači pro výměnu informací pro správné směrování (OSPF, IGRP).

Přidělování portů

Aplikace využívají ke své jednoznačné identifikaci v rámci počítače pro protokol TCP (popřípadě UDP) tzv. porty. Jelikož číslo portu je reprezentováno 16 bitovým číslem, máme k dispozici 65 536 portů. Je dobré vědět, že porty 0 – 1023 jsou rezervovány pro nejběžnější protokoly [16]. Pro úplnost bych ještě dodal, že protokoly využívají pro přenos dat TCP a UDP, ale mnoho z nich je schopných přenos realizovat na obou těchto službách.

Kompletní seznam všech služeb a jim přidělených portů naleznete na http://www.iana.org/assignments/port-numbers. Těmto portům se také říká “well known ports“ (dobře známé porty). Porty 1024 – 49151 se nazývají

registrované porty a jejich použití by mělo být registrováno u organizace ICANN. Zbytek portů je možno použít pro jakoukoliv aplikaci [16].

Vzhledem k tomu, že aplikačních protokolů je nepřeberné množství, nebudu se jednotlivými protokoly nyní zabývat. Pokud budu pracovat s některým aplikačním protokolem, vysvětlím stručně jeho princip a funkci až v dané části mé práce.

2 Sockety

Již v první části své práce jsem se zmiňoval o tom, že socket je jednoznačně definovaný konec komunikačního kanálu. Pokud se na to podíváme obecně, plyne nám z toho, že socket by tedy měl být kombinací IP adresy (ta nám identifikuje počítač) a portu (identifikuje proces, který využívá síťových služeb) [5]. Myslím si, že tuto definici je třeba ještě rozšířit o použitý protokol na transportní vrstvě, z důvodu dvou sad čísel portů, z nichž jedna je pro protokol UDP a druhá pro TCP. Ovšem v samotné implementaci socketů v programovacích jazycích je tomu trochu jinak. Sice stále platí, že socket je přímo propojen s portem, ale toto propojení není automatické. To znamená, že port není přímo parametrem vytvářeného socketu, ale je s ním asociován později. Samotný socket nám definuje pouze to, pomocí jakých služeb budeme aplikační data přenášet. To znamená, že socket je spíše takovým prostředníkem mezi procesem a nižšími vrstvami, umožňuje aplikacím a jejím procesům přistupovat ke službám transportní vrstvy. Sockety v určitých případech umožňují aplikacím přistupovat k ještě k nižším vrstvám síťového modelu. Takovýmto socketům se říká raw sockets (syrové sockety).

Nemyslím si, že by mělo větší smysl se zde o funkcích socketů více zmiňovat. Vše bude mnohem více jasné, až se podíváme na praktické příklady v C#.

2.1 Historie a vývoj

Po implementaci TCP/IP modelu do operačních systémů se objevil problém. Jednotlivé služby jsou sice definovány v RFC, ale nikde není řečeno, jakým způsobem mají aplikace k těmto službám přistupovat. V podstatě to ani

vzhledem k široké škále platforem nebylo možné. Je tedy nutné vytvořit pro každý operační systém aplikační rozhraní, které bude umět k těmto službám přistupovat.

Prvním takovým rozhraním bylo tzv. BSD Socket API vyvinuté pro Unixové systémy BSD na univerzitě v Berkeley kolem roku 1982 (údaje o datu se trochu liší). I když toto rozhraní není z dnešního pohledu ideální (není objektové, obsahuje pouze blokující operace), bylo přelomové. Nejenom že autoři si svůj úkol ještě rozšířili a použili vysokou míru abstrakce, takže je možné toto rozhraní aplikovat nejenom na strukturu TCP/IP, ale i na jiné struktury (AppleTalk, ISO…), ale také naprostá většina implementací socketů do jiných platforem z něj vychází. Nedá se říct, že by byly totožné (liší se syntaxe, práce s objekty atd.), ale princip totožný je [18].

Postupem času se tedy objevovala další rozhraní pro různé platformy, z nich za zmínku stojí určitě Winsock API, které tvořilo rozhraní pro práci se sítí pro systémy Windows.

V některých programovacích jazycích již ani API nepotřebujeme, protože komponenty pro práci se sockety jsou přímo zabudovány v knihovnách (Java, C#). A právě prací se sockety v jazyce C# se budu zabývat dále.

2.2 Sockety v C#

Všechny níže uváděné příklady jsem vytvářel ve Visual Studiu 2008 s nainstalovaným .NET Frameworkem 3.5. Aplikace by ovšem měly fungovat i na starších verzích, jelikož podpora socketů je do .NET Frameworku zařazena již od verze 1.1.

Veškeré nástroje pro práci se sítí nalezneme ve jmenném prostoru System.Net. Nejvíce se nyní budu zabývat jmenným prostorem System.Net.Socket, ovšem využívat budu i jiných tříd, které do něj nepatří, protože bez nich to prostě nejde.

2.2.1 Nejdůležitější třídy pro práci se sockety

V System.Net.Sockets najdeme hned čtyři třídy pro práci se sockety.

Třída Socket

V mé práci jedna z nejdůležitějších tříd. Tato třída reprezentuje objekty, které nám umožňují přistupovat k nižším vrstvám síťového modelu. Jedná se o velmi obecnou třídu, která nám umožňuje manipulovat téměř s jakýmkoliv typem spojení. Daní za to je větší počet parametrů, které je třeba zadat.

Vytvoření nového socketu vypadá následovně:

```
Socket newSocket = Socket(AddressFamily addressFamily,
                          SocketType socketType,
                          ProtocolType protokolType);
```

AddressFamily – určuje s jakým typem adres, bude socket pracovat. V podstatě tím určíme síťový model.

SocketType – určuje způsob přenosu dat.

ProtocolType – protokol, ke kterému budeme pomocí socketu přistupovat.

Třídy TcpClient, TcpListener, UdpClient

Třídy pro zjednodušení práce se sockety. Dle mého názoru jsou trochu zbytečné, neobsahují nic, co bychom nemohli udělat pomocí třídy **Socket**, a naopak zde chybí některé možnosti oproti ní. Přesto je ale pravda, že trochu práce nám mohou ušetřit.

Další třídy

IPEndPoint – objekt této třídy reprezentuje koncový bod. Velmi důležitá třída. S objektem této třídy je třeba Socket asociovat. Parametrem konstruktoru je IP adresa a číslo portu.

IPAddress – objekty této třídy reprezentují konkrétní IP adresu.

2.2.2 Nespojované socktey

O tom, jaké rozdíly jsou mezi spojově orientovanou a nespojovanou službou, jsem již hovořil v první kapitole a nebudu se tím dále zabývat. Pojďme se tedy rovnou podívat, jak takovou jednoduchou nespojovanou službu realizovat.

Ve většině případů jsou síťové aplikace typu klient-server, i když samozřejmě existují výjimky, ale sockety jsou konstruované právě pro takovýto typ komunikace [18]. Vždy tedy budeme muset napsat dvě aplikace, kdy jedna bude typu klient a jedna typu server.

Ještě bych rád upozornil na jednu věc. Všechny aplikace jsou vytvořené jako projekt Windows Form Application. V konzolových aplikacích se může postup trošku lišit.

2.2.2.1 UDP klient

Jak už tedy dávno víme, nespojovaná komunikace je realizována pomocí protokolu UDP.

Vytvořená aplikace bude zasílat zprávu “Hello” pomocí UDP.

V první řadě je zapotřebí si do projektu importovat všechny potřebné knihovny. Budeme tedy potřebovat tyto knihovny:

```
Using System;
using System.Text;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Windows.Forms;
```

Po importu všeho výše uvedeného můžeme začít s prací. Tento program uvedu pro názornost jak za pomoci třídy **Socket,** tak i použitím třídy **UdpClient**.

Se třídou **Socket** by řešení vypadalo následujicím způsobem:

```
private void OdeslatBtn_Click(object sender, EventArgs e)
{
  Socket klient = new Socket(AddressFamily.InterNetwork,
                     SocketType.Dgram, ProtocolType.Udp);
  klient.Connect(textBox1.Text, 8080);
  byte[] zprava = Encoding.ASCII.GetBytes(„Hello“);
  klient.Send(zprava);
}
```

Nejdříve si vytvoříme objekt typu **Socket**, kde zadáme následující parametry:

AddressFamily.InterNetwork – tímto parametrem říkáme, že socket bude pracovat s Internetovým síťovým modelem (TCP/IP), tedy adresovat budeme pomocí IP adres. Je možné zde použít i jiné možnosti namísto *InterNetwork* např. *AppleTaplk*, *IPX* atd., to ale nejsou příliš časté varianty, jimi se ve své práci zabývat nebudu.

SocketType.Dgram – pomocí SocketType určujeme, jakým způsobem budou data přenášena, zvolil jsem datagramový přenos, což je logické vzhledem k použití protokolu UDP.

ProtocolType.Udp – ze socketu budeme přistupovat k protokolu UDP.

Je důležité, aby všechny parametry třídy **Socket** seděly. Mám tím na mysli, že nemůžeme použít „nesmyslnou“ kombinaci, např. strukturu ISO s protokolem TCP.

Následně použijeme metodu *Connect(),* která i když tomu její název napovídá, se nepřipojí k serveru, ale v případě UDP jí pouze zajistíme směr odesílání dat. Prvním parametrem je IP adresa, kterou načítáme z textového pole, a druhým je číslo portu, na kterém server naslouchá. Musíme samozřejmě vědět, kde se server nachází. Já jsem zvolil port 8080. IP adresu bude zadávat uživatel do textového pole.

Poté si vytvoříme pole bytů, které budeme odesílat. Jedná se v podstatě o buffer (zásobník).

V posledním kroku překódovanou zprávu odešleme pomocí metody *Send()*, jejíž parametr je právě pole bytů, ve kterém je zpráva uložena.

Kdybychom chtěli použít třídu **UpdClient**, vypadal by kód následovně:

```
private void OdeslatBtn_Click(object sender, EventArgs e)
{
  UdpClient client = new UdpClient(textBox1.Text,8080);
  byte[] zprava = Encoding.ASCII.GetBytes("Hello");
  client.Send(zprava, zprava.Length);
}
```

Namísto objektu **Socket** vytvoříme objekt typu **UdpClient**. Nejedná se v podstatě o nic jiného, než o jakýsi předpřipravený socket pro správu UDP spojení. Jako parametry můžeme rovnou zadat adresu a port serveru, kam

budou data posílána. Můžeme jej vytvořit i bez parametrů, ale poté bychom museli použít metodu *Connect()*, jako v předcházejícím případě. Zbytek kódu je stejný s tím, že při odeslání zprávy musíme přidat ještě jeden parametr, kterým je délka zprávy.

2.2.2.2 UDP Server

Nyní se podíváme, na tu o něco složitější část práce a tou je server.

Opět musíme importovat výše uváděné knihovny. O tomto se již níže nebudu zmiňovat a budu to pokládat za samozřejmost. Server bude zobrazovat přijatou zprávu a informace o klientu (IP adresu a port), který mu ji zaslal.

Nejdříve opět řešení pomocí třídy **Socket***:*

```
private void Server()
{
    Socket server = new Socket (AddressFamily.InterNetwork,
                                SocketType.Dgram,
                                ProtocolType.Udp);
    IPEndPoint endPoint = new IPEndPoint(IPAddress.Any,
                                         8080);
    server.Bind(endPoint);
    Byte[] data = new Byte[1024];

    while (true)
    {
       EndPoint klientEP = (EndPoint)new
                              IPEndPoint(IPAddress.Any,  0);
       server.ReceiveFrom(data, ref klientEP);
       string prichoziRetezec = Encoding.ASCII.GetString(data);
       listBox1.Items.Add(klientEP.ToString() +"  "+
                          prichoziRetezec);
    }
}
```

Stejným způsobem jako předtím si vytvoříme socket pro protokol UDP. V případě klienta nám bylo jedno, z jakého portu budou data odesílána, a

nechali jsme jej přidělit operačním systémem. V případě severu ale potřebujeme jednoznačně říct, jakému portu bude proces serveru přidělen, aby klient věděl, kam data odeslat. To provedeme tak, že si vytvoříme objekt třídy **IPEndPoint** a konstruktoru přidělíme patřičné parametry. Tedy IP adresu a port. *IPAddres.Any* říká, že je možná jakákoliv IP adresa. V případě, že nás nezajímá port, zadáme jako parametr nulu.

A následně socket s tímto koncovým bodem asociujeme za pomoci metody *Bind().*

V cyklu poté spravujeme jednotlivá příchozí data. Ukončující podmínku si samozřejmě může každý nastavit, jak je potřeba.

Nejpodstatnější metodou je *RecieveFrom(),* která nám naplní, již připravené bytové pole příchozími daty a je metodou blokující. To znamená, že program se na ní zastaví a nebude dále pokračovat, dokud se nevykoná (od toho blokující sockety). Existuje i jiná možnost řešení, než jsou blokující operace, té se ale budu věnovat později.

Ve zbytku cyklu data překódujeme a vypíšeme.

Ovšem takto nám aplikace fungovat nebude. Důvodem je právě již zmiňovaná blokující operace. Program na ní zůstane „viset“ a tím pádem se nám nebude zobrazovat, popřípadě zamrzne okno aplikace. Proto je nutné spustit metodu s blokující operací v novém vlákně.

```
private void Form1_Load(object sender, EventArgs e)
{
    Thread thUDPserver = new Thread(new ThreadStart(Server));
    thUDPserver.IsBackground = true;
    thUDPserver.Start();
}
```

Zde bych upozornil na jednu věc. V případě, že bychom chtěli ukončit aplikaci zavřením okna, nastane problém. Pokud se podíváme do správce úloh, naši aplikaci zde nenajdeme, ovšem pokud se podíváme na spuštěné procesy,

zjistíme, že proces aplikace je stále spuštěn. Toto je dáno naším vytvořeným vláknem, které je stále spuštěné kvůli neukončenému cyklu, a proces tedy stále běží. To lze snadno vyřešit pomocí vlastnosti vláken *IsBackground*, kterou nastavíme na true. Tím řekneme, že vlákno poběží na pozadí a bude automaticky ukončeno, pokud svou práci skončila všechna vlákna na popředí. Implicitně vlákno běží na popředí.

Ovšem ani teď nebude server fungovat tak, jak má. Nyní narazí na problém zejména začátečníci nebo ti, kteří nikdy nepracovali s vlákny. Problém bude v tomto řádku.

```
listBox1.Items.Add(klientEP.ToString() +"  "+ prichoziRetezec);
```

Při prvních přijatých datech nastane výjimka. Problém je v tom, že .NET nepovoluje přistupovat k prvkům GUI z jiného vlákna než z toho, které je vytvořilo. Pokud tomu tak není, jedná se o tzv. Cross-thread operation. Tento problém je třeba nějakým způsobem vyřešit. Řešit to můžeme pomocí delegáta a metody *invoke()*. Nebudu zde podrobně vysvětlovat ani delegáty a metodu *invoke()* (není to předmětem mé práce), ale ukážu pouze řešení.

Nejdříve si vytvoříme delegáta:

```
private delegate void Delegate(string text);
```

Parametry delegáta musí být shodné s parametry metody, na kterou bude delegát ukazovat.

Poté vytvoříme metodu, na kterou bude delegát odkazovat, ta přidá text do seznamu.

```
public void pridejDoSeznamu(string text)
{
    if (listBox1.InvokeRequired)
    {
        Delegate del = new Delegate(pridejDoSeznamu);
        listBox1.Invoke(del, text);
```

```
    }

    else
    {
        listBox1.Items.Add(text);
    }
}
```

A samozřejmě nahradíme v našem vlákně kód pro přidání položky do listboxu voláním této metody:

```
listBox1.Items.Add(klientEP.ToString()+" "+ prichoziRetezec);

pridejDoSeznamu(klientEP.ToString() +"  "+ prichoziRetezec);
```

Nyní by měl být server plně funkční a můžeme obě aplikace (server, klient) vyzkoušet. Do textboxu klienta napíšeme adresu serveru. V případě, že vše zkoušíme na jednom počítači, zadáme do textboxu s adresou slůvko localhost, popřípadě adresu zpětné smyčky 127.0.0.1.

Ještě uvedu druhé řešení pomocí třídy UdpClient.

```
private void Server()
{
    UdpClient server = new UdpClient(8080);
    while (true)
    {
       IPEndPoint endPoint = new IPEndPoint(IPAddress.Any, 0);
       Byte[] prichoziByty = server.Receive(ref endPoint);
      string prichoziRetezec =
                        Encoding.ASCII.GetString(prichoziByty);
      pridejDoSeznamu(prichoziRetezec +"  "
                     +endPoint.ToString());
    }
}
```

Řešení je velice podobné. Hlavní rozdíl je v tom, že nemusíme provádět asociaci socketu s koncovým bodem, ta se provede automaticky při vytvoření objektu, pokud zadáme jako parametr číslo portu. O použití vláken kvůli blokující operaci platí také výše uvedené.

Oba programy (UDPclient a UDP server najdete na přiloženém CD).

2.2.3 Spojově orientované sockety

Spojově orientovaná komunikace bývá zpravidla založena na protokolu TCP a v této části předvedu jak vytvořit TCP klienta a TCP server, který bude schopen realizovat více spojení s klienty najednou.

Nejprve se podíváme na vytvoření TCP klienta

2.2.3.1 TCP klient

```
private void pripoj()
{
    socketServer = new Socket(AddressFamily.InterNetwork,
                              SocketType.Stream,
                              ProtocolType.Tcp);
    IPEndPoint vzdalenyServer = new
                        IPEndPoint(IPAddress.Loopback, 8080);
    try
    {
        socketServer.Connect(vzdalenyServer);
    }
    catch
    {
         MessageBox.Show("Nepodařilo se navázat spojení se
                             serverem");
    }
}
```

Nejprve si opět vytvoříme socket pomocí, kterého se budeme připojovat k serveru (tento socket je instanční proměnnou). Parametry zadáme stejné jako v předcházejícím příkladu s tím rozdílem, že použitý protokol bude TCP. Dále vytvoříme objekt **IPEndPoint**, jehož parametry budou IP adresa a port, na kterém server naslouchá. Poté pomocí metody *Conect()*, navážeme spojení se serverem. Toto spojení je trvalé a bude trvat do doby, než bude jednou ze stran ukončeno (na rozdíl od UDP, kde se spojení nenavazuje). Spojení se serverem

se nemusí z mnoha důvodů podařit navázat, proto je třeba ošetřit výjimku a informovat o neúspěchu uživatele. O tom, kdy metodu zavoláme, si každý může rozhodnut sám (při spuštění aplikace, při stisknutí tlačítka apod.).

Pokud máme navázané spojení, můžeme začít odesílat data na server.

```
private void Odesli()
{
    byte[] odesilanaData =
                    Encoding.ASCII.GetBytes(textBox1.Text);
    try
    {
     socketServer.Send(odesilanaData, odesilanaData.Length, 0);
    }
    catch
    {
       MessageBox.Show("Nelze odeslat zprávu, pravděpodobně
                     nejste připojeni k serveru");
    }
}
```

Nejprve překódujeme zprávu na bytové pole a to odešleme pomocí metody *Send()*, jejímiž dalšími parametry jsou počet bytů, které chceme odeslat, a pozice v poli, od které chceme data odesílat.

Metodu opět zavoláme dle uvážení (např. po stisku klávesy Enter).

2.2.3.2 TCP Server

Nyní se podíváme na, to jak jednotlivá spojení spravovat na straně serveru. Server bude udržovat jednotlivá spojení s klienty, přijímat od nich zprávy.

Opět budeme potřebovat jednu instanční proměnnou:

```
Socket listenerSocket;
```

Již v příkladu s UDP serverem jsme si ukázali, že kvůli blokujícím operacím je třeba vytvářet vlákna. Ani tento příklad nebude výjimkou.

```
private void Form1_Load(object sender, EventArgs e)
{
  Thread naslouchaci = new Thread(new ThreadStart(naslouchej));
   naslouchaci.IsBackground = true;
   naslouchaci.Start();
}
```

Po spuštění aplikace vytvoříme nové vlákno a zavoláme metodu *naslouchej()*, která bude čekat na příchozí požadavky na spojení.

```
private void naslouchej()
{
   listenerSocket = new Socket(AddressFamily.InterNetwork,
                               SocketType.Stream,
                               ProtocolType.Tcp);
   IPEndPoint server = new IPEndPoint(IPAddress.Any, 8080);
   listenerSocket.Bind(server);
   listenerSocket.Listen(10);

    while (true)
    {
        Socket socketKlienta = listenerSocket.Accept();
        if (socketKlienta.Connected)
        {
         PridejKlienta(SocketKlienta.RemoteEndPoint.ToString());
         Thread pracovni = new Thread
                    (new ParameterizedThreadStart(cekejNaData));
         pracovni.IsBackground = true;
         pracovni.Start(socketKlienta);
         }
    }
}
```

Již známým způsobem opět vytvoříme objekty třídy Socket a IPEndPoint a tyto dva objekty sloučíme metodou *Bind()*. Až doposud je vše známé z předcházejících příkladů.

Nyní zavoláme metodu *Listen()*. Tu socket vyžívá pro naslouchaní příchozích pokusů o spojení. Jejím parametrem je maximální počet požadavků,

které mohou čekat na vyřízení. Před zavoláním metody *Listen()* musí být vždy provedena asociace s koncovým bodem metodou *Bind()* [4].

Následně v cyklu (chceme připojit i více klientů) zavoláme další velmi důležitou metodu třídy **Socket** a tou je metoda *Accept()*. Tato metoda nám vrátí socket, který je připojen na koncový bod, jenž je asociován se socketem, pomocí kterého byl požadavek odeslán. Toto je velmi důležitá vlastnost spojově orientovaných socketů. Tato metoda je stejně jako námi již známé *Send()* a *Receive()* blokující.

Dále si zobrazíme klienta v listBoxu (opět si musíme dát pozor na křížení vláken).

Nakonec vytvoříme nové vlákno, ve kterém budeme přijímat data zaslaná daným klientem (pro každé připojení vlákno). Toto je nutnost vzhledem k tomu, že v aktuálním vlákně je blokující metoda *Accept()*, navíc pro příjem dat budeme také využívat blokující metodu *Receive()*.

Metoda spuštěná v novém vlákně by mohla vypadat takto:

```
private void cekejNaData(object socket)
{
  bool n = true;
  Socket aktualniSocket = (Socket)socket;
  string text;
  byte[] buffer = new byte[1024];
  while (n == true)
  {
       try
       {
           aktualniSocket.Receive(buffer);
       }
       catch
       {
           n = false;
           lock (listBox1)
           {
       OdstranSocket(aktualniSocket.RemoteEndPoint.ToString());
           }
           aktualniSocket.Close();
```

```
        }
        text = Encoding.ASCII.GetString(buffer);
        zobrazText(text);
        Array.Clear(buffer, 0, 1024);
    }
}
```

Socket, pomocí kterého budeme přijímat příchozí zprávy, získáme z parametru metody. Ten je typu **object** z toho důvodu, že delegát **ParameterizedThreadStart** předání jiného parametru nepovoluje [4]. Dále k příjmu dat použijeme metodu *Receive()*. Jedná se opět o blokující operaci. S touto metodou jsme se již setkali u nespojované komunikace. V případě TCP plní v podstatě stejnou funkci, tedy příjem dat, jen způsob použití se trochu liší. V tomto případě nám naplní bytové pole, které zadáme jako její parametr. Pokud je spojení mezi klientem a serverem z nějakého důvodu přerušeno, projeví se to výjimkou právě na tomto řádku. Toho se dá velmi snadno využít a v bloku catch ošetříme vše potřebné (seznam přihlášených klient apod.). Pokud vše proběhne v pořádku, server vypíše příchozí zprávu do textového pole.

Jak jste si jistě všimli, příklady aplikací na spojovanou komunikaci se funkčně v podstatě neliší od příkladu s protokolem UDP. Výhoda socketů nad protokolem TCP se projeví až ve složitějších aplikacích. Klíčem je metoda *Accept()*, která nám vrátí socket vytvořený pro komunikaci s jedním klientem. Pokud máme tento socket k dispozici, jsme schopni již realizovat stálou, obousměrnou a spolehlivou komunikaci (samozřejmě je třeba tomu server i klienta přizpůsobit, např. dopsáním metody pro příjem dat u klienta apod.) nebo zrealizovat komunikaci mezi jednotlivými klienty.

A právě aplikace pro komunikaci mezi jednotlivými klienty jsem zařadil jako jeden z ukázkových příkladů. Server udržuje seznam přihlášených klientů a informuje každého klienta o ostatních přihlášených klientech. Klient poté může zaslat zprávu jakémukoliv z přihlášených klientů. Zpráva nejdříve přijde

na server s informací o tom, jakému klientovi má zprávu přeposlat, a poté je doručena adresátovi.

Ukázky kódu v mé práci jsem zjednodušil tak, abych demonstroval pouze to nejpodstatnější, popisovat zde celé programy ChatServer a ChatClient by bylo zbytečné.

Stejně jako u UDP i pro protokol TCP existují třídy pro jednodušší práci se sockety. Jsou to třídy **TcpClient** a **TcpListener**. Tyto třídy reprezentují jakési předpřipravené vzory socketů pro spojení pomocí TCP a usnadňují nám jejich implementaci [1].

V těchto příkladech jsme viděli všechny nejdůležitější metody a třídy socket a jejich použití. Níže v tabulce je jejich souhrn.

Accept()	Vytvoří socket pro nově vytvořené spojení
Bind()	Asociuje socket s místním koncovým bodem s (objektem IPEndPoint)
Close()	Ukončí spojení socketu a uvolní s ním spojené prostředky
Connect()	Sestaví spojení se vzdáleným hostitelem
Listen()	Nastaví socket do stavu naslouchání (před touto metodou musí být vždy zavolána metda Bind)
Receive()	Přijímá data a naplní jimi buffer
ReceiveFrom()	Obdoba metody Receive pro datagramovou službu. Kromě naplnění bufferu uloží vzdálený EndPoint
Send()	Odešle data připojenému socketu
SendTo()	Odešle data na daný koncový bod.

Tabulka 1:Přehled metod pro práci se sockety

2.2.4 Asynchronní sockety

Až doposud jsme využívali synchronních socketů. Synchronní operace obecně vykonávají jednotlivé instrukce postupně a žádná z následujících instrukcí kódu se nezačne vykonávat, dokud nebude ta předcházející dokončena. Stejně je tomu i u socketů. V případě synchronních socketů se setkáme s tzv. blokujícími operacemi (někdy jsou synchronní sockety nazývány blokujícími sockety). Blokující operace, se kterými jsme setkali, jsou: *Accept(),Connect(), Receive(), ReceiveFrom(), Send(), SendTo()*.

V případě těchto operací není synchronní přístup úplně ideální, a to z toho důvodu, že může trvat velmi dlouhou dobu, než budou vykonány, a teprve poté začne program vykonávat další příkazy. Program se v tuto chvíli stává prakticky nepoužitelným. Příkladem může být nějaký komunikační klient. Pokud v této aplikaci zavoláme metodu *Receive()* pro příjem dat, program musí čekat, než tato akce bude vykonána. To ovšem může být otázkou minut či hodin, protože na klienta musí být nejdříve nějaká zpráva zaslána. A po celou tuto dobu nemůže klient vykonávat žádné jiné operace (aktualizace GUI, odeslání zprávy atd.). Jedním z možných řešení této situace, které jsme doteď používali, je využití vláken. Každou blokující operaci spustíme ve vlastním vlákně, kterým jsou střídavě přidělovány výpočetní prostředky.

Existuje ovšem i jiné řešení a tím je použití asynchronních metod (od nich tzv. asynchronní sockety). Princip myšlenky asynchronních operací spočívá v tom, že si zažádáme o vykonání operace, program mezitím pokračuje a vykonává jiné části kódů, a teprve až bude asynchronní operace dokončena, vyzvedneme si její výsledek a ten podle potřeby zpracujeme. Svým způsobem se asynchronní operce od použití vláken tolik neliší, protože při zavolání asynchronní metody je vytvořeno taktéž vlákno (typu démon) [19], ve kterém se metoda zpracovává. Ke zpracování výsledků využíváme tzv. callback

metody. Tato metoda se zavolá po dokončení asynchronní operace. Nejprve je ale potřeba se ještě seznámit s jednou třídou.

2.2.4.1 AsyncCallBack

Objekty třídy **AsyncCallBack** jsou speciální typy delegatů, které se využívají pro zpracování výsledků asynchronních operací. Parametrem těchto objektů je metoda, která zpracovává výsledek asynchronní operace. V souvislosti s touto třídou je třeba si představit ještě rozhraní **IAsyncResult**. Toto rozhraní představuje výsledek asynchronního volání a je parametrem metody, na kterou delegát odkazuje [4]. Funkce obou tříd budou lépe vidět na příkladech, které si ukážeme později.

2.2.4.2 Použití asynchronních metod

Každá z blokujících metod má svůj asynchronní ekvivalent. Asynchronní metody se skládají ze dvou částí zpravidla začínajících Begin a End. Část Begin spustí asynchronní metodu a jedním z jejích parametrů je asynchronní delegát odkazující na metodu pro zpracování dat po ukončení operace.

Část End slouží pro dokončení asynchronní metody volané funkce [1].

V tabulce níže jsou dvojice jednotlivých asynchronních metod.

BeginAccept(),	EndAccept()
BeginConnect()	EndConnect()
BeginReceive(),BeginReceiveFrom()	EndReceive(),EndReceiveFrom()
BeginSend(),BeginSendTo()	EndSend(),EndSendTo()

Tabulka 2: Přehled asynchronních metod pro práci se sockety

Nyní se již budu zabývat příklady použití asynchronních metod. Pro ukázku jsem vytvořil funkčně stejné programy jako již známé ChatClient a ChatServer s tím, že místo blokujících operací byly použity asynchronní metody. Tyto dva programy jsou opět přiloženy na CD pod názvy ChatClienAsync a ChatServerAsync. Stejně jako u blokující verze programy velmi zjednoduším a funkčně zkrátím na pouhé připojení k serveru a odeslání zprávy na server, který zobrazí klienty a poslední zaslanou zprávu.

2.2.4.3 Klient

Připojení – BeginConnect() a EndConnect()

Připojení k serveru bude první věcí, kterou musí náš klient umět. Řekněme, že klient se pokusí k serveru připojit ihned po svém spuštění.

```
private void Form1_Load(object sender, EventArgs e)
{
   socketServer = new Socket(AddressFamily.InterNetwork,
                             SocketType.Stream,
                             ProtocolType.Tcp);
   IPEndPoint vzdalenyServer = new
                     IPEndPoint(IPAddress.Loopback, 8080);
   socketServer.BeginConnect(vzdalenyServer,new
                            AsyncCallback(Pripoj),null);
   }
```

Obvyklým způsobem vytvoříme socket a objekt **IPEndPoint**, který reprezentuje vzdálený server. Následně zavoláme metodu *BeginConnect(),* kde prvním parametrem bude koncový bod serveru. Druhým parametrem bude objekt třídy **AsyncCallBack,** jehož parametrem bude metoda, jejíž kód se vykoná, když bude vše připraveno k dokončení [1]. Posledním parametrem je objekt typu **object**. Ten slouží k předání objektu, který si můžeme vyzvednout

pomocí vlastnosti *AsyncState* třídy **IAsyncResult**. Často se stává, že žádný objekt předávat nepotřebujeme, tak zadáme hodnotu null.

V tuto chvíli musíme ještě napsat metodu *Pripoj()*.

```
private void Pripoj(IAsyncResult result)
{
    try
    {
      socketServer.EndConnect(result);
      pripojeno = true;
    }
    catch
    {
      MessageBox.Show("Nepodařilo se navázat spojení se
                       serverem");
    }
}
```

Parametrem každé metody delegáta **AsyncCallBack** je objekt třídy **IAsyncResult**, který reprezentuje výsledek námi požadované asynchronní operace [4]. Zavoláním metody *EndConnet()* (parametrem je právě výše zmíněný parametr metody), dokončíme připojení, a v případě neúspěchu ošetříme výjimku. Pokud vše proběhne v pořádku, klient se úspěšně připojil.

Odesání zprávy – BeginSend() a EndSend()

Odesílání krátkých textových zpráv pomocí asynchronních metod je popravdě řečeno zbytečné, protože toto odeslání proběhne téměř okamžitě a vlákno tedy nijak neblokuje, ale v jiných případech může být asynchronní odesílání užitečné.

```
private void Odeslat_Click(object sender, EventArgs e)
{
    byte[] buffer = Encoding.ASCII.GetBytes(textBox1.Text);
    try
    {
```

```
      socketServer.BeginSend(buffer, 0, buffer.Length,
                             SocketFlags.None,
                             new AsyncCallback(Odesli), null);
    }
    catch
    {
      MessageBox.Show("Nelze odeslat zprávu, pravděpodobně
                       nejste připojeni k serveru");
    }
}
```

Odeslání dat provedeme pomocí metody *BeginSend()*. Prvním parametrem je pole bytů, které chceme odeslat. Druhý je tzv. offset, je typu integer a určuje index prvního bytu. Parametr **SocketFlags** určuje zvláštní vlastnosi odesílaných paketů [4]. S parametrem **AsyncCallBack** jsme se již setkali, tentokrát delegát odkazuje na metodu *Odesli()*. Posledním parametrem je objekt, který můžeme předat do výsledku reprezentovaného rozhraním

IAsyncResult.

Callback metoda bude tentokrát velmi jednoduchá.

```
private void Odesli(IAsyncResult result)
{
      int odeslano = socketServer.EndSend(result);
}
```

Metoda *EndSend()* má jeden parametr result, který reprezentuje výsledek asynchronní operace spuštěné metodou *BeginSend()*, dokončí odesílání dat a vrátí počet odeslaných bytů.

2.2.4.4 Server

Pojďme se tedy podívat jak to bude vypadat na straně serveru. Zde se seznámíme s asynchronními verzemi metod *Accept()* a *Recive()*.

```
private void Naslouchej()
{
   listenerSocket = new Socket(AddressFamily.InterNetwork,
                               SocketType.Stream,
                               ProtocolType.Tcp);
   IPEndPoint server = new IPEndPoint(IPAddress.Any, 8080);
   listenerSocket.Bind(server);
   listenerSocket.Listen(10);
   listenerSocket.BeginAccept(new AsyncCallback(Akceptuj),null);
}
```

Opět zavoláme námi již známou trojici metod s tím, že místo metody *Accept()* použijme metodu *BeginAccept()*. Prvním parametrem Je asynchronní delegát a druhým opět objekt, který si chceme předat do metody volané delegátem. Žádný objekt předávat nepotřebujeme (tedy v tom případě, pokud je objekt listenerSocket vytvořen jako instanční proměnná).

```
private void Akceptuj(IAsyncResult result)
{
   Socket socketKlienta = listenerSocket.EndAccept(result);
   PridejSocket(socketKlienta.RemoteEndPoint.ToString());
   byte[] buffer = new byte[1024];

   socketKlienta.BeginReceive(buffer, 0, buffer.Length,
                              SocketFlags.None,
                              new AsyncCallback(PrijemDat),
                              new object[]{socketKlienta,buffer});
listenerSocket.BeginAccept(new  AsyncCallback(Akceptuj),  null);
}
```

Metoda *EndAccept()* dokončí operaci započatou *BeginAccept()* a vrátí socket pro komunikaci s novým klientem.

V tuto chvíli již můžeme na socketu očekávat data od klienta a zavoláme tedy metodu *BeginRecieve()*. Parametry, které obsahuje, jsem již popisoval. Tentokrát ale budeme potřebovat metodě delegáta předat objekty, a to hned dva (socket vrácený metodou *EndAccept()* a buffer, který má být naplněn daty). Předat můžeme jen jeden objekt, který je ovšem typu **object**, takže do něj můžeme uložit prakticky cokoliv. Když tedy potřebujeme předat více objektů,

jednoduše si vytvoříme pole, do kterého je uložíme, a toto pole předáme jako parametr volané metodě.

A nyní se již setkáme s opravdovou podstatou a významem asynchronních socketů. Aniž bychom čekali na dokončení metody *BeginRecive()*, pokračujeme dále. Vzhledem k tomu, že chceme server pro více klientů, zavoláme znovu metodu *BeginAccept()* pro příjem dalšího požadavku na spojení.

A nakonec metoda pro příjem dat od klienta:

```
private void PrijemDat(IAsyncResult result)
{
  bool pripojeno = true;
  object[] obj = (object[])result.AsyncState;
  Socket aktualniKlient = (Socket)obj[0];
  byte[] buffer = (byte[])obj[1];
  try
  {
    int prijato = aktualniKlient.EndReceive(result);
  }
  catch
  {
     OdstranSocket(aktualniKlient.RemoteEndPoint.ToString());
      pripojeno = false;
   }
   String text = Encoding.ASCII.GetString(buffer);
   zobrazText(text);
   Array.Clear(buffer, 0, 1024);

   if (pripojeno == true)
   {
     aktualniKlient.BeginReceive(buffer, 0, buffer.Length,
                                 SocketFlags.None, new
                                 AsyncCallback(PrijemDat),
                     new object[] { aktualniKlient, buffer });
   }
}
```

Získání objektu předávaného z asynchronní metody provedeme pomocí vlastnosti **AsyncSate**, kde je tento objekt uložen. Je typu **object** proto je

potřeba ho přetypovat na správný typ. Poté zavoláme metodu *EndRecieve()*, která vrací počet přijatých bytů. Pokud by spojení z nějakých důvodů selhalo, musíme ošetřit výjimku a smazat informace o připojeném klientovi ze seznamu. Pokud vše proběhne v pořádku a spojení je stále navázáno, znovu zavoláme metodu *BeginReceive()*, abychom mohli přijímat další zprávy zaslané tímto klientem.

2.2.4.5 Asynchronní vs. synchronní socket

Krása asynchronních socketů spočívá v tom, že jsme schopni na jednom vlákně zpracovávat najednou více spojení. To, že vše probíhá na jednom vlákně, ovšem není až tak úplně pravda. Asynchronní metody využívají tzv. **ThreadPool** . Jedná se v podstatě o zásobník vláken. Ten má k dispozici určitý počet vláken a ty přiděluje jednotlivým asynchronním operacím. Programátorovi tedy odpadá problém s vytvářením vláken a jejich správou. Toto vše udělá systém za programátora a ten se tedy může plně věnovat logice programu.

Nic ale není zadarmo. Asynchronní metody jsou o něco složitější na pochopení a výsledný kód mi přijde méně přehledný než v případě vláken. Je to právě dané callback metodami, kterými si v podstatě rozhodíme jednu operaci do dvou metod. V případě jednoduchých aplikací není tento rozdíl markantní, ale při složitějších už by mohl být znát.

Nejpodstatnějším problémem s asynchronními metodami se mi ale jeví synchronizace (je to logické vzhledem k jejich názvu). U asynchronních metod těžko určíme, kdy se operace dokončí. V podstatě tím, že je použijeme, říkáme, že je nám to jedno. Ale co když nám to jedno není? Existuje mnoho případů, kdy potřebujeme dokončit jednu akci, než začneme druhou. Určitě by se i tento problém dal vyřešit, ale v těchto případech je nejlepší se asynchronním metodám vyhnout.

2.2.5 Přenos souborů

2.2.5.1 Třída NetworkStream

Streaming pomáhá programátorům přesunout větší množství dat. Třída **NetworkStream** poskytuje socketům rozhraní pro práci se streamy [1]. Tuto třídu nalezneme ve jmenném prostoru **System.Net.Sockets**.

Vytvořit objekt této třídy můžeme více způsoby. Buďto použijeme konstruktor, jehož parametrem je socket, pro který chceme stream vytvořit.

```
NetworkStream streamSocketu = new NetworkStream(Socket);
```

Pokud používáme třídu TcpClient, zavoláme metodu *GetStream()*, která vrací objekt třídy **NetworkStream**.

```
NetworkStream streamSocketu = tcpklinent.GetStream();
```

Nyní si prakticky předvedeme, jak se třídou **NetworkStream** pracovat. Opět vytvoříme dvě aplikace (klienta a server). Pomocí klienta bude možné nahrát soubor na server.

Odeslání souboru – klient

```
private void Form1_Load(object sender, EventArgs e)
{
  server = new Socket(AddressFamily.InterNetwork,
                      SocketType.Stream,
                      ProtocolType.Tcp);
   try
   {
     server.Connect(IPAddress.Loopback, 8080);
   }
  catch
   {
     MessageBox.Show("nepodařilo se navázat spojení se
                      serverem");
   }
}
```

Po spuštění klienta se klasickým způsobem připojíme k serveru.

Poté potřebujeme nalézt soubor, který chceme odeslat.

```
private void najit_Click(object sender, EventArgs e)
{
  OpenFileDialog open = new OpenFileDialog();
  open.ShowDialog();
  textBox1.Text = open.FileName;
}
```

Vytvoříme objekt třídy **OpenFileDialog**, který představuje klasické dialogové okno pro vyhledání souboru, ten zobrazíme a název souboru vypíšeme do textového pole.

Následující metoda odešle soubor.

```
private void odesli_Click_1(object sender, EventArgs e)
{
  Stream streamSouboru = null;
  try
  {
    streamSouboru = File.OpenRead(textBox1.Text);
    Thread odeslani = new Thread(new
                         ParameterizedThreadStart(Odesli));
    odeslani.IsBackground = true;
    odeslani.Start(streamSouboru);
  }
  catch (Exception vyjimka)
  {
     MessageBox.Show(vyjimka.Message);
  }
}

public void Odesli(object stream)
{
  OdesliNazevAVelikost();
  Stream streamSouboru = (Stream)stream;
  NetworkStream sitovyStream = null;
  try
  {
     NetworkStream sitovyStream = new NetworkStream(server);
  }
  catch(Exception ex)
  {
```

```
      MessageBox.Show(ex.Message);
   }
   byte[] bufferSouboru = new byte[1024];
   int kOdeslani;

   while (true)
   {
      kOdeslani = streamSouboru.Read(bufferSouboru, 0, 1024);
      if (kOdeslani == 0)
      { break; }
      else
      {
         sitovyStream.Write(bufferSouboru, 0, 1024);
      }
    }
    streamSouboru.Close();
    sitovyStream.Close();
}
```

Nejprve musíme vytvořit stream souboru. To provedeme zavoláním metody *OpenRead()* třídy **File**, kde parametrem je soubor, respektive cesta k němu a ta vrátí objekt třídy **Stream**. Samotné odesílání může trvat delší dobu, tak budeme raději odesílat v jiném vlákně.

Pokud vše proběhne v pořádku, zavoláme metodu, která odešle název a velikost souboru na server. K této metodě se ještě vrátím později.

Zavoláním metody *Read()* uložíme načtené hodnoty ze streamu do námi připraveného pole. Následně vytvoříme objekt třídy **NetworkStream**, jejímž parametrem je síťový přístup představovaný socketem, pro který budeme stream poskytovat a zavoláme metodu *Write(),* kde prvním parametrem je pole bytů, ze kterého chceme data do síťového streamu zapisovat. Dalšími parametry jsou informace, o tom odkud a kolik bytů v poli chceme zapsat. Není vhodné celý soubor načíst do pole najednou, proto je lepší soubor odesílat např. po kilobytech, protože v případě větších souborů bychom vytvořili příliš velkou zátěž pro operační paměť.

Nakonec ještě oba námi vytvořené streamy zavřeme.

Nyní se podíváme na metodu pro odeslání názvu souboru. Proč vůbec takovou metodu psát a ztěžovat si práci? Důvody mohou být dva. Na server chceme uložit soubor pod stejným názvem, pod jakým ho máme uložený na disku. Druhý důvod (a dle mého názoru ještě podstatnější) je, že na server chceme ukládat více než jeden typ souboru, a proto je nezbytné server informovat alespoň o příponě souboru. Dlouho jsem přemýšlel, jak tento problém vyřešit. Nakonec mě napadlo, že název souboru na server můžeme zaslat nezávisle na přenášených datech.

Nejprve je tedy nutné zavolat metodu pro odeslání názvu a velikosti ve výše popisované metodě pro odeslání dat.

Samotná metoda bude vypadat následovně:

```
private void OdesliNazev()
{
FileInfo info = new FileInfo(textBox1.Text);
string[] cesta = textBox1.Text.Split('\\');
byte[] buffer = Encoding.ASCII.GetBytes(cesta[cesta.Length –
                              1]+"|"+info.Length.ToString());

try
  {
     server.Send(buffer);
  }
  catch
  {
      MessageBox.Show("soubor nelze odeslat");
  }
}
```

Jednoduše si pomocí metody *Split()* z cesty k souboru „vytáhneme“ její název a metodou *Send()* ho odšleme s velikostí souboru získanou pomocí třídy **FileInfo**. Proč odesíláme i velikost souboru, se dozvíme níže.

Řešení je vcelku banální, jenom na něj přijít.

Tímto bychom měli hotového klienta.

2.2.5.2 Příjem a uložení souboru – server

Po spuštění aplikace vytvoříme nové vlákno a v něm spustíme metodu pro naslouchání příchozích požadavků na spojení.

```
private void Form1_Load(object sender, EventArgs e)
{
Thread naslouchajici = new Thread(new ThreadStart(Naslouchej));
naslouchajici.IsBackground = true;
naslouchajici.Start();
}

private void Naslouchej()
{
  Socket listenerSocket = new
                      Socket(AddressFamily.InterNetwork,
                      SocketType.Stream,ProtocolType.Tcp);
   IPEndPoint ipep = new IPEndPoint(IPAddress.Any,8080);
   listenerSocket.Bind(ipep);
   listenerSocket.Listen(10);
   while (true)
   {
     Socket socketKlienta = listenerSocket.Accept();
     if (socketKlienta.Connected)
     {

pridejDoSeznamu(socketKlienta.RemoteEndPoint.ToString() + "
                 prihlasen");
         Thread pracovni = new Thread(new
         ParameterizedThreadStart(PrijemDat));
          pracovni.IsBackground = true;
          pracovni.Start(socketKlienta);
        }
     }
   }
```

V těchto dvou metodách se nesetkáváme s ničím novým. Po přijetí spojení získáme socket pro komunikaci s klientem. Spustíme nové vlákno, ve kterém zavoláme metodu, která bude tentokrát přijímat zasílaný soubor z klienta a ukládat jej.

```
private void PrijemDat(object socket)
{
  Socket socketKlienta = (Socket)socket;
 NetworkStream streamZeSite = new NetworkStream(socketKlienta);
  byte[] buffer = new byte[1024];
  string[] nazevAvelikost =
                        NazevSouboruAVelikost(socketKlienta);
  string nazev = nazevAvelikost[0];
  int velikost = Convert.ToInt32(nazevAvelikost[1]);
  int prijato = 0;
  int nactenaData;

  if (nazev != null)
  {
    Stream streamDoSouboru = File.OpenWrite(nazev);
    while (prijato < velikost)
    {
      if (streamZeSite.DataAvailable == true)
      {
         try
         {
           nactenaData = streamZeSite.Read(buffer, 0, 1024);
           streamDoSouboru.Write(buffer, 0, 1024);
           prijato = prijato + nactenaData;
         }
         catch(Exception ex)
         {
            MessageBox.Show(ex.Message);
         }
       }
     }
     streamDoSouboru.Close();
     streamZeSite.Close();
     pridejDoSeznamu("Soubor Zapsán");
     PrijemDat(socketKlienta);
  }
}
```

Jako první vytvoříme socket, který jsme předali jako parametr vláknu. Ale může být pouze typu **object**, tak jej musíme přetypovat. Nyní již můžeme vytvořit objekt **NetworkStream** pro tento socket. V dalším kroku je zapotřebí zjistit název souboru a jeho velikost. Pro tento účel si opět vytvoříme samostatnou metodu, kterou ukážu později. Pokud byl název metodou

v pořádku vrácen, začneme psát v podstatě opačný proces tomu, který jsme použili na straně klienta.

Vytvoříme stream pro zápis do souboru zavoláním metody *OpenWrite()* třídy **File**.

Začneme číst data síťového streamu pomocí metody *Read()*, která dělá opak toho co třída *Write()*. Data načtená ze streamu zapíše do pole, které je prvním parametrem metody, další dva parametry jsou shodné jako u metody *Write()*. Poté nastreamujeme data z pole do souboru. Tento proces pomocí cyklu opakujeme, dokud bude počet přijatých bytů menší než počet, o kterém nás informoval klient. Podmínkou v cyklu zjišťujeme, zda jsou na streamu dostupná data. To zjistíme pomocí vlastnosti třídy **NetworkStream** a tou je *DataAvailable*. Hodnotu true má pokud jsou na streamu dostupná data ke čtení.

Nakonec zavřeme oba používané streamy a zavoláme znovu metodu pro příjem dat, pokud by klient chtěl zaslat další soubor.

Takto uložená data se nám budou ukládat do složky, kde je uložen spouštěcí soubor serveru (relativní cesta). Pokud bychom jej chtěli uložit jinam, jednoduše před název souboru připojíme řetězec reprezentující cestu k námi vybrané složce.

Nyní se ještě podíváme na metodu pro přijetí názvu a velikosti souboru.

```
private string[] NazevSouboruAVelikost(Socket socketKlienta)
{
  byte[] buffer = new byte[1024];
  string[] nazevAVelikost = new string[2];
  try
  {
    socketKlienta.Receive(buffer);
    string prichozi = Encoding.ASCII.GetString(buffer);
    nazevAVelikost = prichozi.Split('|');
  }
  catch
  {
    pridejDoSeznamu(socketKlienta.RemoteEndPoint.ToString() +
```

```
                    "  odhlasen");
  }
  return nazevAVelikost;
}
```

Již známým způsobem použijeme metodu *Receive()*. Zde je důležité použít blokující verzi této metody. Vždy budeme potřebovat, aby metoda *NazevSouboru()* proběhla dříve, než budeme pokračovat v metodě *PrijemDat()*. Kdybychom použili asynchronní metody, řízení by bylo předáno zpět metodě *PrijemDat()* a v řádku, kde vytváříme stream do souboru, bychom neznali název souboru, který chceme uložit.

Pokud vše proběhne v pořádku, zpětně překódujeme zprávu. A vrátíme výsledek, který je názvem souboru.

Ukázky kódu jsem použil z programů, které jsem nazval TCPclient a TCPserver.

Stejně jako třída **Socket** i třída **NetworkStream** disponuje asynchronními metodami pro odesílání a příjem dat a těmi jsou metody *BeginRead()*, *EndRead()*, *BeginWrite()* a *EndWrite()* [4].

Další možností jak odesílat soubory je metoda *SendFile()*. Tato metoda ovšem nemá svůj přijímací ekvivalent. Proto si celý příjem dat neboli protokol musíme zřídit sami a data přijímat klasicky metodou *Receive()*.

2.2.6 Služby nižších vrstev

Ve většině implementací socketů je umožněn i přístup k nižším vrstvám síťového modelu, než je vrstva transportní. Slouží k tomu tzv. sockety typu "raw"(syrové).

Za jejich pomoci můžou aplikace přistupovat ke službám i nižších vrstev, než je vrstva transportní. Zejména ke službám třetí vrstvy (IP, ICMP), ale

v některých implementacích je možné přistupovat i níže k linkové vrstvě. Tyto sockety užitečné zejména při implementací takových aplikací, jako jsou routery nebo monitory síťového provozu [18].

2.2.6.1 ICMP

Nejčastějším případem přístupu k nižším vrstvám pomocí raw socketů je přístup k protokolu ICMP. Jeho funkci jsem zmiňoval v teoretické části.

Běžný uživatel se nejčastěji setká s protokolem ICMP v programu ping, kterým testujeme konektivitu s cílovým bodem (adresou). K tomuto testu se využívají dva typy ICMP paketů: „Echo Request“ (žádost o echo) a „Echo Reply“ (odpověď na echo). Ačkoli teď nebudu příliš originální, tak právě na takovém příkladu předvedu práci s raw sockety. Příklady s programem ping, které lze najít, mi přijdou zbytečně složité.

Velmi důležitou věcí je, že raw sockety nikterak neformátují data. To je dost podstatný problém. To znamená, že si musíme tento paket vytvořit sami ručně. [1].

Na štěstí hlavička ICMP protokolu nepatří mezi nejobsáhlejší. Nicméně je velmi důležité, aby jednotlivé části hlavičky byly zadány přesně, jinak by je vzdálený hostitel nemohl přečíst. Ovšem není problém si za hlavičku (do datové části) dopsat svoje vlastní údaje.

Hlavičku ICMP jsem již uváděl v teoretické části. Pojďme se tedy podívat, jak ji vytvořit v našem programu.

```
public byte[] vytvorPacket()
{
  byte[] zprava = Encoding.ASCII.GetBytes("pokus");
  byte[] packet = new byte[zprava.Length + 5];
  byte[] sum = BitConverter.GetBytes((Int16)0);
```

```
   packet[0] = 8;
   packet[1] = 0;
   Array.Copy(sum, 0, packet, 2, sum.Length);
   Array.Copy(zprava, 0, packet, 4, zprava.Length);

  byte[] kontrolniS = BitConverter.GetBytes(getChecksum(packet,
                                              4 + zprava.Length));
   Array.Copy(kontrolniS, 0, packet, 2, kontrolniS.Length);
   return packet;
}
```

První tři řádky obsahují deklaraci potřebných polí. Prvním je samotná zpráva, tedy obsah datové části paketu. Druhým je pole představující celý paket a posledním je dvoubytové pole, které představuje kontrolní součet. Poté do pole reprezentujícího paket začneme vkládat jednotlivé části. První byte paketu určuje jeho typ. Druhý byte je tzv. kód. Ten také určuje, o jaký druh ICMP paketu se jedná (jde o jakési jemnější dělení) [3]. Kombinace 8 a 0 určuje paket s žádostí o echo. Dále vložíme kontrolní součet, který je prozatím nulový, a nakonec samotnou zprávu. Z takto vytvořeného paketu musíme vypočítat kontrolní součet a ten nahradíme za dříve vloženou nulu. Je velmi důležité, aby byl kontrolní součet spočten, tak jak má být. V opačném případě by cílový hostitel považoval paket za chybný a jednoduše by ho zahodil.

Výpočet kontrolního součtu je poměrně složitou záležitostí a nesouvisí úplně s tématem socketů, tak jsem si do svého programu metodu pro jeho výpočet vypůjčil z knihy C# network programming.

```
public Int16 getChecksum(byte[] data,int delka)
{
   Int32 chcksm = 0;
   int packetsize = data.Length;
   int index = 0;
    while ( index < delka)
    {
   chcksm += Convert.ToInt32(BitConverter.ToInt16(data, index));
      index += 2;
    }
    chcksm = (chcksm >> 16) + (chcksm & 0xffff);
```

```
    chcksm += (chcksm >> 16);
    return (Int16)(~chcksm);
}
```

[1]

Pokud máme vytvoření paketu hotové, můžeme jej odeslat.

```
private void button1_Click(object sender, EventArgs e)
{
  Socket socket = new Socket(AddressFamily.InterNetwork,
                             SocketType.Raw,
                             ProtocolType.Icmp);
  try
  {
   IPEndPoint cil = new IPEndPoint
                (Dns.Resolve(textBox1.Text).AddressList[0], 0);
   byte[] packet = vytvorPacket();
   socket.SendTo(packet, packet.Length, SocketFlags.None, cil);
   cekejNaOdpoved((EndPoint)cil);
  }
  catch(Exception vyjimka)
  {
    MessageBox.Show(vyjimka.Message);
  }
}
```

V prvním řádku metody vytváříme raw socket. To provedeme tak, že typ socketu nastavíme na Raw a typ protokolu zvolíme Icmp. Poté z názvu cílového hostitele získáme pomocí třídy **Dns** jeho IP adresu a výše uváděnou metodou vytvoříme paket. Vzhledem k tomu, že ICMP je službou nespojovanou, musíme pro odeslání použít metodu *SendTo()*. Poté zavoláme metodu, která bude čekat na odpověď na echo.

```
public void cekejNaOdpoved(EndPoint cil)
{
  byte[] odpoved = new byte[1024];
  socket.SetSocketOption(SocketOptionLevel.Socket,
                         SocketOptionName.ReceiveTimeout, 5000);
  try
  {
    socket.ReceiveFrom(odpoved, ref cil);
    byte kod = odpoved[20];
```

```
     byte typ = odpoved[21];
     Int16 soucet = BitConverter.ToInt16(odpoved, 22);
     VypisPrichoziPacket(cil.ToString() + "   " + kod.ToString()
          + "   " + typ.ToString() + "   " + soucet.ToString());
    }
    catch
    {
       MessageBox.Show("vypršel časový limit pro odpověď");
    }
}
```

Zde stojí za zmínku dvě věci. První z nich je metoda *SetSocketOption()*, pomocí které můžeme nastavit různé vlastnosti socketu. Zde nastavíme časový limit pro přijetí paketu, protože odpověď nemusí dorazit a v tom případě bychom na ni čekali marně. Limit pěti sekund je plně dostačující a může být i menší.

Druhou věcí je získání dat z příchozího paketu. Vzhledem k tomu, že ICMP paket je v IP paketu, tedy za jeho záhlavím, které má 20 bytů musíme data číst až od položky s indexovým číslem 20. Pokud vše proběhne v pořádku, program vypíše IP adresu, ze které byla odpověď přijata a data ze záhlaví paketu. Jedná-li se o odpověď na echo, bude typ i kód paketu 0.

Ještě doplním, že takto psát program pro testování konektivity je poměrně zbytečné. .NET Framework totiž disponuje třídami, které jsou nástroji přímo pro práci s echy (Ping, PingReply). Ty se nacházejí ve jmenném prostoru System.Net.NetworkInformation. Pouze jsem chtěl na známém příkladu předvést funkci raw socketů.

Toto byla poslední část tykající se socketů a programování na nižších vrstvách. Poslední část mé práce se bude týkat práce s aplikačními protokoly v C#.

3 Aplikační protokoly v C#

Aplikačních protokolů existuje obrovské množství a to se stále rozrůstá. Není prakticky možné napsat zde třeba jen větu o každém z nich. Proto se zaměřím pouze na tři protokoly a těmi bude SMTP, IMAP a FTP. Vybral jsem právě tyto tři, protože se jedná o klasické a známé protokoly, ale zároveň mají dost odlišnou funkci.

Dále si v této kapitole také představíme další možnou variantu vývoje síťových aplikací a tou je balík nástrojů pro vývoj síťových aplikací s v jazyce C# od firmy Rebex.

3.1 FTP (File Transfer Protocol)

FTP je protokolem sloužícím k přenosu dat mezi počítači a je jedním z nejstarších protokolů TCP/IP. Má mnoho funkcí (umožňuje řízený přístup, výpis vzdáleného adresáře, specifikaci formátů atd.). Tento protokol běží na portu 20 a 21 a jeho úplnou specifikaci si můžeme přečíst na RFC 959 [20].

3.1.1 FTP v C#

Pokud chceme v .NET Frameworku využívat ve své aplikaci protokol FTP, máme více možností. První z nich je využití tříd, které nám nabízí samotný .NET Framework. Ovšem nástroje pro tyto účely nejsou úplně ideální, ale pro základní funkce postačí. Další způsob je napsání si vlastní knihovny pro FTP, ale vzhledem k tomu, že se jedná o velmi starý a často využívaný protokol, je více než pravděpodobné, že to už někdo udělal. A tím se dostávám k posledním dvěma variantám. Jednou z nich je, že zabrousíme po internetu a najdeme si nějakou knihovnu zdarma. V tomto případě se spíše jedná o klasickou třídu, která definuje jednotlivé funkce FTP. K nalezení jich je opravdu dost.

A konečně poslední varianta. Zakoupení profesionální knihovny (popřípadě celého balíku knihoven). Jedním z takových balíků je Rebex Total pack. Níže se budu zabývat právě tímto balíkem společně se základními nástroji .NET Frameworku.

3.1.2 FTP klient

Nyní se tedy podíváme jak vytvořit FTP klienta. Abychom mohli lépe porovnat oba způsoby, napsal jsem dva funkčně totožné klienty a u každé funkce si vždy předvedeme oba způsoby.

FTP klient bude umět všechny základní funkce, jako je nahrání a stažení souboru, vytvoření složky na serveru, procházení složek a smazání složky nebo souboru. Interakci s uživatelem budeme provádět pomocí prvku ListView, do kterého budeme zobrazovat obsah aktuální složky. A uživatel bude moci dvojklikem přecházet mezi složkami.

Vše potřebné v .NET Frameworku, najdeme ve třídách **FtpWebResponse**, **FtpWebRequest** a **WebRequestMethod.Ftp**. Co se týče Rebexu, je potřeba přidat do referencí v solution exploreru odkaz na knihovnu pro práci s FTP, konkrétně se jedná o soubor **Rebex.Net.Ftp.dll** [23]. Také si musíme importovat jednotlivé jmenné prostory do našich dvou projektů. V případě .NET budeme používat stejné jmenné prostory jako u socketů. Při použití nástrojů Rebexu importujeme jmenný prostor Rebex.Net.

3.1.2.1 Přihlášení k serveru

.NET Framework

V tomto případě žádné přihlášení neprobíhá. Každý požadavek serveru musíme zasílat zároveň s přihlašujícími údaji. Ovšem pro první zobrazení obsahu serveru jsem vytvořil metodu, která přihlášení simuluje.

```
private void Prihlasit_Click(object sender, EventArgs e)
{
   adresa = "ftp://" + uri.Text;
   ZobrazObsah();
}
```

V prvním řádku nastavíme hodnotu instanční proměnné *adresa*. Tu budeme brát z textového pole a bude ji zadávat uživatel. Poté zavoláme metodu, pro získání a zobrazení obsahu serveru, kterou si předvedeme později.

Rebex

Zde již připojování a přihlašování probíhá. Je to výhodné, protože později nás to ušetří otravného nastavování přihlašovacích údajů každému požadavku. Nejprve si ale vytvoříme instanční proměnnou

```
private static Ftp klient = new Ftp();
```

Jedná se o objekt typu **Ftp**, jejíž metody reprezentují jednotlivé funkce, které u FTP využíváme a jak uvidíte níže, její použití je velice jednoduché.

Nyní tedy metoda pro přihlášení.

```
private void Prihlasit_Click(object sender, EventArgs e)
{
  try
  {
    klient.Connect(uri.Text);
    klient.Login(uzivatel.Text, heslo.Text);
    zobrazObsah();
  }
  catch (Exception exc)
  {
    MessageBox.Show(exc.Message);
  }
}
```

Pomocí metody *Connect()* se připojíme k serveru. Jejím parametrem je adresa serveru. Výhodou je, že začátek adresy (ftp://) si doplní sama. Metoda

Login() přihlásí uživatele k serveru. Parametry jsou uživatelské jméno a heslo. Ty opět zadá uživatel do textových polí. Nakonec také zavoláme metodu pro zjištění obsahu serveru.

3.1.2.2 Zjištění obashu serveru

Zde už je práce trochu náročnější. Nejde ani tak o samotné získání jednotlivých položek ze serveru, ale spíše nastane problém, pokud je chceme nějakým rozumným způsobem zobrazit. Zejména v případě základních nástrojů .NET Frameworku si programátor opravdu „užije“. Data do ListView budeme zobrazovat do sloupců, kde první sloupec bude obsahovat název položky, druhý příponu (jedná-li se o soubor) a třetí velikost položky.

.Net Framework

```
private void ZobrazObsah()
{
  List<string> obsah = new List<string>();
  StreamReader stream;
  try
  {
    FtpWebRequest pozadavek =
                 (FtpWebRequest)FtpWebRequest.Create(adresa);
    pozadavek.Credentials =
              new NetworkCredential(uzivatel.Text, heslo.Text);
    pozadavek.Method =
                    WebRequestMethods.Ftp.ListDirectoryDetails;
    WebResponse odpoved = pozadavek.GetResponse();
    stream = new StreamReader(odpoved.GetResponseStream());
    while (!stream.EndOfStream)
    {
       obsah.Add(stream.ReadLine());
    }
    obsah.RemoveAt(0);
  }
  catch(Exception ex)
  {
     MessageBox.Show(ex.Message);
  }
  smazListView();
```

```
  foreach (string polozka in obsah)
  {
    string[] udaje = Udaje(polozka);
    if (JeSoubor(polozka))
    {
     string[] nazevApr = rozdelNazev(udaje[0]);
     pridejDoListView(new string[]
                    {nazevApr[0],nazevApr[1],udaje[1]});
    }
    else
    {
      pridejDoListView(new string[] { udaje[0], "Složka",
                       udaje[1] });
     }
  }
}
```

Nejprve vytvoříme nový požadavek pomocí statické metody třídy **FtpWebRequest** *Create()*. Jejím parametrem je adresa FTP serveru. Dále nastavíme vlastnost *Credentials*, která reprezentuje přihlašovací údaje a to tak, že jí přiřadíme objekt typu **NetworkCredential,** jehož parametry jsou uživatelské jméno a heslo. Dále nastavíme vlastnost *Method*, kterou v podstatě říkáme, o jaký typ požadavku se jedná. Jednotlivé metody protokolu FTP jsou obsaženy ve třídě **WebRequestMethods.Ftp** a v tomto případě zvolíme *ListDirectoryDetails*. Následně získáme odpověď serveru, kterou reprezentuje objekt třídy **FtpWebresponse**. Z něho získáme stream dat a pomocí cyklu while a objektu StreamReader je přečteme. Zde narážíme na problém, protože informace o každé položce získáme jako textový řetězec, což značně ztěžuje následné zjištění námi požadovaných údajů. Pro tento účel musíme ještě dopsat metodu, kterou jsem nazval *Udaje()*. Dále si také sami musíme dopsat metodu pro rozpoznání složky a souboru. Ani jedno v případě Rebexu není nutné. Tyto metody zde ukazovat nebudu, ale prohlédnout si je můžete v přiloženém programu Ftpclient.

Rebex

Nyní konečně uvidíme, jak nám nástroje Rebexu značně ulehčují práci

```
private void zobrazObsah()
{
  FtpList seznam = new FtpList();
  try
  {
    seznam = klient.GetList();
  }
  catch(Exception ex)
  {
     MessageBox.Show(ex.Message);
  }
  smazListView();
  pridejDoListView(new string[] { ".." });
  foreach (FtpItem i in seznam)
  {
    if (i.IsFile)
    {
      string[] nazev = rozdelNazev(i.Name);
      pridejDoListView(new string[] { nazev[0], nazev[1],
                       i.Size.ToString() });
    }
    else
    {
      pridejDoListView(new string[] { i.Name, "Složka",
                       i.Size.ToString() });
    }
  }
}
```

Zde vidíte, že celé získání obsahu serveru zrealizujeme pomocí jedné metody *GetList()*. Ta nám vrátí objekt typu **FtpList**, který obsahuje objekty **FtpItem** reprezentující jednotlivé položky serveru. To má obrovskou výhodu, protože jednotlivé údaje získáme pomocí vlastností objektů **FtpItem** (Size, Name, IsFile atd.).

3.1.2.3 Upload

Jelikož nahrávání souboru může trvat dlouhou dobu, je více než vhodné níže uváděné metody spouštět ve vlastním vlákně.

.Net Framework

```
private void Nahraj(object obj)
{
  string nazev = (string)obj;
  FileInfo info = new FileInfo(nazev);
  try
  {
    FtpWebRequest pozadavek =
                   (FtpWebRequest)FtpWebRequest.Create(adresa +
                    "/" + info.Name);
    pozadavek.Credentials = new
                  NetworkCredential(uzivatel.Text, heslo.Text);
    pozadavek.Method = WebRequestMethods.Ftp.UploadFile;
    FileStream stream = File.OpenRead(nazev);
    Stream streamServeru = pozadavek.GetRequestStream();
    byte[] buffer = new byte[1024];
    int precteno = 1;
    while (precteno > 0)
    {
      precteno = stream.Read(buffer, 0, buffer.Length);
      streamServeru.Write(buffer, 0, buffer.Length);
    }
       stream.Close();
       streamServeru.Close();
     }
     catch (Exception ex)
     {
        MessageBox.Show(ex.Message);
     }
     ZobrazObsah();
}
```

Stejným způsobem jako v předcházejícím případě vytvoříme požadavek s tím, že tentokrát k adrese serveru přidáme ještě název souboru, aby cesta byla kompletní. Vlastnost *Method* v tomto případě nastavíme na metodu protokolu *UploadFile*. Posléze musíme vytvořit dva streamy. Jeden pro čtení dat ze

souboru a druhý pro zápis na server, který získáme pomocí metody *GetrequestStream()*, a pomocí cyklu data zapsaná do bufferu jedním streamem zapíšeme tím druhým na server. Opět doporučuji, aby data byla nahrávána tímto způsobem a nebyl do bufferu načten celý soubor najednou kvůli zbytečnému zaplnění paměti. Oba streamy zavřeme a znovu zobrazíme obsah serveru.

Rebex

```
private void Nahraj(object obj)
{
   string nazev = (string)obj;
   FileInfo info = new FileInfo(nazev);
   try
   {
     klient.PutFile(nazev, klient.GetCurrentDirectory() + "/" +
                    info.Name);
   }
   catch (Exception ex)
   {
      MessageBox.Show(ex.Message);
   }
      zobrazObsah();
}
```

Už na první pohled je jasné, že i zde nám Třída **Ftp** značně usnadní práci. Celá proces nahrání souboru na server se skrývá v metodě *PutFile()*, jíž předáme jako parametr cestu k souboru na lokálním počítači a cílovou adresu na serveru. Zbytek udělá za nás. Jednodušší už to snad ani být nemůže.

3.1.2.4 Vytvoření složky

Nejprve musíme vyřešit způsob, jakým bude uživatel zadávat název složky. To uděláme tak, že vytvoříme další formulář, který bude obsahovat textové pole, do kterého uživatel zadá název složky. Tento formulář se zobrazí, když uživatel dá příkaz k vytvoření složky. Po stisknutí tlačítka OK se zavolá

metoda hlavního formuláře pro vytvoření složky na serveru a ta bude vypadat následovně.

.NET Framework

```
public void vytvorSlozku(String nazev)
{
    try
    {
     FtpWebRequest pozadavek =
                   (FtpWebRequest)FtpWebRequest.Create(adresa +
                    "/" + nazev);
     pozadavek.Credentials = new
                  NetworkCredential(uzivatel.Text, heslo.Text);
     pozadavek.Method = WebRequestMethods.Ftp.MakeDirectory;
     FtpWebResponse resp =
                  (FtpWebResponse)pozadavek.GetResponse();
     }
     catch (Exception ex)
     {
       MessageBox.Show(ex.Message);
     }
     ZobrazObsah();
}
```

Jedná se opět o sestavení FTP požadavku. Metodu protokolu zvolíme *MakeDirectory*.

Rebex

```
public void PridejSlozku(string nazevSlozky)
{
  string soucasna = klient.GetCurrentDirectory();
  try
  {
    klient.CreateDirectory(soucasna+"/"+ nazevSlozky);
    }
    catch(Exception ex)
   {
     MessageBox.Show(ex.Message);
   }
   zobrazObsah();
}
```

Stejně jako v předchozích případech je všechen kód tykající se FTP nahrazen jedinou metodou. Pro vytvoření složky je to *CreateDirectory()*. Dále zde vidíme ještě jednu užitečnou metodu třídy **Ftp**, kterou je *GetCurrentDirectory()* vracející současnou složku, v níž se nacházíme. V prvním případě si naši současnou pozici musíme hlídat sami.

3.1.2.5 Shrnutí

Nemyslím si, že by bylo třeba předvádět zde další funkce FTP (jedná se o principiálně totožné příklady). Další metody pro stažení souboru, smazání složky či souboru nebo přechod mezi složkami si můžete prohlédnout v programech, ze kterých pocházely výše uvedené příklady. Jedná se o programy Ftpclient a FtpRebex.

Jak jsme mohli vidět, Rebex nám v případě FTP určitě hodně usnadní práci. V podstatě pro každou funkci je připravena metoda, kterou jednoduše zavoláme, a o nic dalšího se nemusíme starat. Ovšem výše uvedeným výhody Rebexu nekončí. Obsahuje opravdu širokou škálu dalších nástrojů. Za zmínku stojí např. nástroje pro šifrované verze FTP (SFTP a FTPS), přenos většího množství souborů a složek, práci s proxy, přenos s kompresí atd. Také nám poskytuje asynchronní verze metod.

3.2 SMTP a IMAP

SMTP (Simple Mail Transfer Protocol) je protokolem pro odesílání elektronické pošty a jeho původní specifikace je k nalezení v RFC 821. Komunikace probíhá tak, že SMTP klient odešle zprávu na SMTP server a ten zjistí, jestli je cílový adresát jeho lokálním uživatelem. Pokud tomu tak není, předá pomocí svého SMTP klienta poštu dalšímu serveru. V případě, že adresát

je lokálním uživatelem serveru, bude pošta uložena do jeho poštovní schránky. Protokol SMTP je ale tzv. jednocestný protokol. Je tedy možné pomocí něho poštu odesílat, ale už ji není možné přijmout, respektive ji vyzvednout z poštovní schránky [21]. Pro tyto účely slouží jiné protokoly a těmi jsou POP3 a IMAP. Ve své aplikaci budeme používat protokol IMAP, a tak se zaměřím na něj. Jedná se tedy o protokol pro vzdálený přístup k poštovní schránce. K tomu slouží i POP3, ale mezi oběma protokoly existuje několik zásadních rozdílů. POP3 je mnohem jednodušší službou. Pokud náš klient používá IMAP, tak je k serveru připojen po celou dobu jeho spuštění, zatímco u POP3 je klient přihlášen pouze po dobu vyřizování požadavku (stažení zprávy apod.). IMAP nám nabízí také širší možnosti vzdálené správy (práce se strukturou adresářů), také umožňuje přihlášení více klientů najednou. V současné době se nejvíce využívá verze IMAP4, která je specifikována v RFC 3501 [22]. Některé poštovní servery mohou se svými klienty preferovat vlastní protokol, ale všechny servery zahrnují i podporu těchto standardních protokolů.

3.2.1 Přihlášení a šifrování

Než začneme programovat samotnou aplikaci, je třeba si říci několik vět o šifrování a autorizaci při práci s poštovními protokoly.

To že IMAP servery vyžadují autorizaci, je zcela pochopitelné. Do své schránky má přístup pouze jeho majitel, který se prokáže uživatelským jménem (tím zpravidla bývá jeho emailová adresa) a heslem. Co se týče SMTP serverů, dříve autorizaci nevyžadovaly. V současné době ovšem drtivá většina těchto serverů přihlašovací údaje vyžaduje, zejména kvůli spamovacím robotům. Tyto údaje jsou shodné s přihlašovacími údaji k emailové schránce. Proto vždy, když budeme chtít odeslat email, budeme muset sestavit přihlášení s těmito údaji.

Nešifrovaná verze SMTP využívá port 25 a IMAP port 143. V současné době je ale nešifrovaný přístup k SMTP serveru i k IMAP serveru většinou blokován. To ovšem není nikterak velký problém. K těmto protokolům existují šifrované varianty SMTP/SSL a IMAP/SSL. SSL (Security Socket Layer) je vrstva vložená, do síťového modelu TCP/IP mezi vrstvu aplikační a transportní z důvodu chybějícího standardu pro zabezpečení přenášených dat. Poskytuje zabezpečení jak na autorizační úrovni (zejména směrem ze serveru ke klientovi), tak také zahrnuje celou řadu šifrovacích algoritmů. V současné době se používají protokoly verze SSL 3.0 a TLS 1.0 popřípadě 1.1. TLS (Transport Layer Security) není jiným protokolem, ale pouze novější verzí SSL. Ještě zbývá dodat, že šifrované verze poštovních protokolů běží na portu 465 nebo 587 (SMTP) a 993(IMAP).

3.2.2 SMTP a IMAP v C#

Pokud budeme chtít pracovat s těmito protokoly pomocí základních nástrojů .NET Frameworku, uspějeme pouze napůl. .NET Framework sice disponuje třídami pro odeslání emailů (**SmtpClient**, **MailAddress** a **MailMessage**). Nutno podotknout, že práce s těmito třídami je velmi příjemná. Ovšem pokud budeme hledat nástroje pro práci s protokolem IMAP, narazíme, protože .NET Framework žádné nemá. To znamená, že pokud budeme chtít s tímto protokolem, musíme použít jiné řešení. Například stáhnout knihovnu, kterou již někdo naprogramoval, nebo si vytvořit svou vlastní pomocí socketů. My se v další a poslední části podíváme opět na řešení pomocí balíku Rebex.

3.2.3 Emailový klient

Naše ukázková aplikace tedy bude jednoduchý emailový klient. Bude se umět přihlásit ke své poštovní schránce, odesílat zprávy, zobrazovat jak

seznam přijatých zpráv, tak celou zprávu, mazat zprávy a pracovat s přílohami (nahrání, uložení).

Opět musíme přidat reference na knihovny do našeho projektu. Jedná se soubory **Rebex.Mail.dll**, **Rebex.Net.Imap.dll**, **Rebex.Net.SecureSocket.dll**, **Rebex.Net.Smtp.dll** [23]. Také musíme importovat potřebné jmenné prostory do projektu.

```
using Rebex.Net;
using Rebex.Mail;
```

Jako první věc budeme potřebovat přihlášení k serveru. Pro zadání parametrů připojení využijeme nový formulář, který zobrazíme po přihlášení tlačítka.

```
private void btnPrihlasit_Click(object sender, EventArgs e)
{
   Prihlaseni pr = new Prihlaseni(this);
   pr.Show();
}
```

Formuláři předáme jako parametr třídu současného formuláře, abychom mohli využívat jeho metod. Samotná metoda pro nastavení parametrů ve druhém formuláři bude vypadat takto:

```
private void button1_Click(object sender, EventArgs e)
{
  string imapServer = textBox4.Text;
  int imapPort = Convert.ToInt32(textBox6.Text);
  string smtpServer = textBox3.Text;
  int smtpPort = Convert.ToInt32(textBox5.Text);
  string email = textBox1.Text;
  string heslo = textBox2.Text;
  form.Prihlas(imapServer, imapPort, email, heslo, smtpServer,
               smtpPort);
  form.Prijate();
  this.Close();
}
```

Jednoduše přiřadíme jednotlivé hodnoty z textových polí formuláře proměnným, které poté předáme metodě hlavního formuláře *Prihlas()*. Proměnná *form* musí být instanciována v konstruktoru této třídy.

A nyní konečně samotná metoda pro přihlášení k IMAP serveru.

```
public void Prihlas(string imapServer, int imapPort, string
          email, string heslo, string smtpServer, int smtpPort)
{
  this.smtpServer = smtpServer;
  this.imapServer = imapServer;
  this.email = email;
  this.heslo = heslo;
  this.smtpPort = smtpPort;
  this.imapPort = imapPort;
  klient = new Imap();
  TlsParameters parametry = new TlsParameters();
  parametry.Version = TlsVersion.Any;
  try
  {
    klient.Connect(imapServer, imapPort, parametry,
                   ImapSecurity.Implicit);
    klient.Login(email, heslo);
    lbUzivatel.Text = email;
    btnPrihlasit.Visible = false;
   }
   catch(Exception ex)
   {
     MessageBox.Show(ex.Message);
   }
}
```

Instančním proměnným (budeme je potřebovat i v jiných metodách) představujícím jednotlivé parametry připojení přiřadíme hodnoty předaných parametrů. Další instanční proměnné typu **Imap** přiřadíme nový objekt. Právě tato třída nám poskytuje řadu metod pro komunikaci pomocí protokolu IMAP. Dále ještě musíme nastavit šifrování. K tomu vyžijeme třídu **TlsParameters** a její vlastnost *Version*, kterou reprezentuje objekt typu **TlsVersion**. Jelikož nám je vcelku jedno, která verze šifrovacího protokolu bude použita, můžeme ji

nastavit na *Any*. Následují asi dvě nejdůležitější metody pro připojení. Použitím metody *Connect()* se připojíme k serveru. Další metoda *Login()* přihlásí uživatele k jeho účtu.

Další věcí, kterou by měl asi každý poštovní klient umět, je zobrazit seznam přijatých zpráv. I pro tohle nám Rebex nabízí metodu. Pro zobrazení seznamu využijeme stejně jako u FTP ListView, kterému opět nejdříve musíme nastavit vlastnost *Details*.

```
listView1.View = View.Details;
```

Samotná metoda pak bude vypadat následovně.

```
public void Prijate()
{
   listView1.Items.Clear();
   try
   {
     klient.SelectFolder("Inbox");
     seznamZprav = klient.GetMessageList
                  (ImapListFields.FullHeaders);
     foreach (ImapMessageInfo zprava in seznamZprav)
     {
       listView1.Items.Add(new ListViewItem(new string[] {
                        zprava.SequenceNumber.ToString(),
                        zprava.From.ToString(), zprava.Subject,
                        zprava.Date.ToString() }));
     }
   }
   catch(Exception ex)
   {
     MessageBox.Show(ex.Message);
   }
}
```

Jak jsem již říkal, v případě IMAP protokolu je možné (spíše i nutné) pracovat s adresářovou strukturou. Pomocí metody *SelectFolder()* vybereme aktuální složku. Bývá nepsaným pravidlem, že veškerá příchozí pošta je ve schránce uložena ve složce Inbox. Třída **Imap** nám nabízí i další metody pro práci se složkami, ty ovšem potřebovat nebudeme. Co ale určitě potřebovat

budeme, je metoda *GetMessageList()*. Ta vrací objekt typu **ImapMessageCollection**, který udržuje seznam objektů **ImapMessageInfo** [23], pomocí jejichž vlastností získáme potřebné informace (předmět, od koho byla zpráva přijata atd.) o zprávách a za pomoci cyklu je vypíšeme do ListView.

Nyní když máme už seznam jednotlivých zpráv, můžeme si jednotlivé zprávy zobrazit. Celou zprávu ve svém programu zobrazuji do nové záložky. K tomu jsem využil prvek GUI TabControl. Zobrazovat zprávu budeme po dvojkliku na ni v ListView.

```
private void listView1_DoubleClick(object sender, EventArgs e)
{
  int cislo = Convert.ToInt32(listView1.SelectedItems[0].Text);
  ZobrazitZpravu(cislo);
}
```

Jednou z informací, které do seznamu zobrazujeme, je pořadové číslo zprávy a právě podle tohoto čísla budeme rozpoznávat zprávu, kterou chceme zobrazit.

```
public void ZobrazitZpravu(int cisloZpravy)
{
  klient.SelectFolder("Inbox");
  MailMessage zprava = klient.GetMailMessage(cisloZpravy);
  lbOd.Text = tabPage2.Text = zprava.From.ToString();
  lbPredmet.Text = zprava.Subject.ToString();
  tbTextZpravy.Text = zprava.BodyText.ToString();

  foreach (Attachment priloha in zprava.Attachments)
  {
    lboxPrilohy.Items.Add(priloha.FileName);
  }
  aktualneZobrazena = zprava;
}
```

Opět nastavíme aktuální složku. Poté pomocí *GetMailMessage()* získáme zprávu a následně vypíšeme jednotlivé informace, jako je předmět nebo text

zprávy. Co se týče příloh, jsou uloženy v seznamu a jsou představovány objekty **Attachment**. Velmi jednoduše tedy vypíšeme jejich názvy do ListBoxu.

Nyní se podíváme, jak přílohu ze schránky stáhnout.

```
private void btnPrilohy_Click(object sender, EventArgs e)
{
 if (lboxPrilohy.SelectedItem != null)
 {
   SaveFileDialog dialog = new SaveFileDialog();
   dialog.FileName = lboxPrilohy.SelectedItem.ToString();
   dialog.ShowDialog();

  foreach (Attachment priloha in aktualneZobrazena.Attachments)
  {
   if (priloha.FileName == lboxPrilohy.SelectedItem.ToString())
   {
      priloha.Save(dialog.FileName);
   }
  }
 }
 else
 {
   MessageBox.Show("Vyberte přílohu, kterou chcete uložit");
 }
}
```

Vše je opět velmi jednoduché. Po vybrání přílohy ze seznamu a stisknutí tlačítka se uživateli zobrazí dialog, pomocí kterého zvolí, kam chce přílohu uložit. Pro samotné stažení máme opět připravenu metodu. Konkrétně se jedná o metodu třídy **Attachment** *Save()*, jejímž parametrem je název přílohy.

Co se týče protokolu IMAP, program obsahuje ještě jednu metodu pro smazání zprávy.

```
private void btnSmazat_Click(object sender, EventArgs e)
{
  klient.SelectFolder("Inbox");

  foreach (ListViewItem polozka in listView1.SelectedItems)
```

```
  {
    klient.DeleteMessage(Convert.ToInt32(polozka.Text));
  }
  Prijate();
}
```

K tomu už snad ani není co dodat. Snad jen, že parametrem metody *DeleteMessage()* je opět pořadové číslo zprávy.

Poslední věcí, kterou uděláme, je odeslaní zprávy. Pro tu si opět vytvoříme novou záložku v prvku TabControl, na kterou umístíme ostatní grafické prvky.

```
private void btnOdeslat_Click(object sender, EventArgs e)
{
  Smtp klient = new Smtp();
  TlsParameters parametry = new TlsParameters();
  parametry.Version = TlsVersion.Any;
  try
  {
    klient.Connect(smtpServer, smtpPort, parametry,
                   SmtpSecurity.Implicit);
    klient.Login(email, heslo);
    MailMessage zprava = new MailMessage();
    zprava.To = tbKomu.Text;
    zprava.Subject = tbPredmet.Text;
    zprava.BodyText = tbTextOdeslane.Text;
    zprava.From = email;
    foreach (string priloha in lboxPridane.Items)
    {
      Attachment pr = new Attachment(priloha);
      zprava.Attachments.Add(pr);
    }
    klient.Send(zprava);
  }
  catch (Exception ex)
  {
    MessageBox.Show(ex.Message);
  }
}
```

Klíčovou třídou pro odeslání zprávy je **Smpt**. Připojení a přihlášení je stejné jako u třídy **Imap**. Posléze vytvoříme samotnou zprávu (třída **MailMessage**) a nastavíme jednotlivé parametry. V cyklu vytvoříme přílohy z názvů, které jsou v ListBoxu. Ty jsem nechal uživatele přidávat pomocí tlačítka, po jehož

stisknutí se objeví dialogové okno, ve kterém uživatel vyhledá soubor, který chce přiložit. Ukazovat zde tuto metodu mi přijde zbytečné. Nakonec zprávu odešleme pomocí *Send()*.

Nástroje pro SMTP obsahuje i samotný .NET Framework. Co je ovšem zajímavé, že třídy se velmi podobají těm, kterými disponuje Rebex.

```
private void btnOdeslat_Click(object sender, EventArgs e)
{
  try
  {
    SmtpClient klient = new SmtpClient(smtpServer, smtpPort);
    klient.EnableSsl = true;
    klient.Credentials = new NetworkCredential(email, heslo);
    MailMessage zprava = new MailMessage(email,tbKomu.Text);
    zprava.Subject = tbPredmet.Text;
    zprava.Body = tbTextOdeslane.Text;

    foreach (string priloha in lboxPridane.Items)
    {
      Attachment pr = new Attachment(priloha);
      zprava.Attachments.Add(pr);
    }
    klient.Send(zprava);
  }
  catch (Exception ex)
  {
     MessageBox.Show(ex.Message);
  }
}
```

Jak můžeme vidět, celá metoda je téměř totožná s tou předcházející. Třída **SmtpClient** až na pár drobností odpovídá třídě **Smtp** ve Rebexu. Stejně je tomu tak i u tříd **MailMessage**. Jak jsem již říkal, jedná bohužel se o osamocený případ, co se týká tříd pro aplikační protokoly v .NET Frameworku.

Chtěl bych ještě upozornit, že jsem využíval pouze 30 denní zkušební verzi knihoven Rebexu. To znamená, že programy se po měsíci stanou nefunkčními. Je tedy nutné po této době odstranit reference na tyto knihovny a vložit do něj nové.

4 Závěr

Myslím, že na vytvořených aplikacích je vidět, že pokud chceme pracovat ve svých programech s nižšími vrstvami, než je vrstva aplikační, mohou být sockety velmi vhodnou volbou. V kombinaci s jazykem C# se stávají silným nástrojem pro vytváření vlastní komunikace a protokolů mezi jednotlivými aplikacemi. Jsme pomocí nich schopni přistupovat k mnoha službám a využívat je v podstatě pro všechny druhy síťových aplikací. Navíc implementace socketů v .NET Frameworku je objektová, takže i když se jedná o poměrně starou technologii, dokáže držet krok s moderními trendy. Dalším kladem je možnost využití asynchronních metod a existence doplňujících tříd pro práci se sockety a pro programování síťových aplikací obecně.

Co se týče aplikačních protokolů, je na tom .NET Framework o něco hůře. I když jsme si ukázali, že disponuje některými třídami (např. **SmtpClient**), se kterými se pracuje velmi dobře, tak většinou je opak pravdou. Zdaleka ne vždy je práce s jeho třídami ideální a někdy není z důvodů absence tříd pro některé protokoly (např. IMAP) vůbec možná.

Při práci s FTP jsme mohli vidět, že i když je .NET Framework vybaven třídami pro manipulaci s tímto protokolem, určitě se nejedná o nejjednodušší řešení. K tomuto mi výborně posloužil balík nástrojů pro .NET Framework od firmy Rebex. Už při pouhém pohledu na kód jednotlivých metod je vidět, že nám tento balík ušetří mnoho práce. Je to bezpochyby velmi užitečný nástroj a je jedním z možných řešení problémů, na které v jazyce C# při práci s aplikačními protokoly narazíme.

Co se týče cílů mojí práce, myslím, že se mi je podařilo splnit. Byla vytvořena sbírka ukázkových a okomentovaných programů, které jsem se následně pokusil co nejpodrobněji popsat. Tento popis jsem rozšířil o popis nebo alespoň naznačení řešení problémů se související problematikou. Aplikace se z výše zmiňovaným popisem, tvoří kompletního průvodce

programováním síťových aplikací, kterého mohou využít především začátečníci. Dále jsem také svou práci rozšířil o ukázku alternativní možnosti při programování síťových aplikací v jazyce C#, kterou je balík Rebex Total Pack.

Reference

1. BLUM, Richard. *C# Network Programming*. [s.l.] : Sybex, 2003. 647 s. ISBN 0782141765.

2. REID, Fiach. *Network Programming in .NET*. [s.l.] : Digital Press, 2004. 541 s. ISBN 1-55558-315-6.

3. DOSTÁLEK, Libor, KABELOVÁ, Alena. *Velký průvodce protokoly TCP/IP a systémem DNS*. [s.l.] : [s.n.], 2000. 426 s. ISBN 80-7226-323-4.

4. Http://msdn.microsoft.com/cs-cz/library/default.aspx [online]. c2010 [cit. 2010-01-16]. Dostupný z WWW: <http://msdn.microsoft.com/cs-cz/library/default.aspx>.

5. DRAXLER, Petr. *Datové sítě*. [s.l.] : [s.n.], [200?]. 57 s.

6. Protokol (informatika) In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, 12. 1. 2006, 28. 4. 2009 [cit. 2009-20-12]. Dostupné z WWW: <http://cs.wikipedia.org/wiki/Protokol_%28informatika%29>.

7. PETERKA, Jiří. Referenční model ISO/OSI. *eArchiv.cz* [online]. 1999, [cit. 2009-12-20]. Dostupný z WWW: <http://www.earchiv.cz/anovinky/ai1552.php3>.

8. Protokol TCP - I.. *Http://pc-site.owebu.cz/* [online]. 2000, 0, [cit. 2010-01-02]. Dostupný z WWW: <http://pc-site.owebu.cz/?page=PTCP>.

9. PETERKA, Jiří. Aplikační vrstva TCP/IP. *EArchiv.cz* [online]. 1993, [cit. 2009-12-25]. Dostupný z WWW: <http://www.earchiv.cz/a93/a319c110.php3>.

10. BARTOŠEK, Miroslav. Krátce z historie Internetu. *ÚVT MU* [online]. 1995, [cit. 2010-01-02]. Dostupný z WWW: <http://www.ics.muni.cz/zpravodaj/articles/22.html>.

11. RFC 792. *RFC792 - Internet Control Message Protocol*. RFC : J. Postel, 1981. 20 s. Dostupné z WWW: <http://www.faqs.org/rfcs/rfc792.html>.

12. *RFC 793*. California : Information Sciences Institute University of Southern California 4676 Admiralty Way Marina del Rey, California 90291, 1981. 85 s. Dostupné z WWW: <http://www.faqs.org/rfcs/rfc793.htm>.

13. *RFC 768*. Information Sciences Institute : J. Postel, 1980. 3 s. Dostupné z WWW: <http://www.faqs.org/rfcs/rfc768.html>.

14. INTERNET PROTOCOL. California : Information Sciences Institute University of Southern California 4676 Admiralty Way Marina del Rey, California 90291, 1981. 45 s. Dostupné z WWW: <http://www.ietf.org/rfc/rfc791.txt>.

15. Čtyři vrstvy TCP/IP. *Http://pc-site.owebu.cz/* [online]. 2000, [cit. 2010-01-15]. Dostupný z WWW: <http://pc-site.owebu.cz/?page=PTCPIP1>.

16. Seznam %C4%8D%C3%ADsel port%C5%AF TCP a UDP In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, 18.6. 2006, 25.2. 2010 [cit. 2010-02-8]. Dostupné z WWW: <http://cs.wikipedia.org/wiki/Seznam_%C4%8D%C3%ADsel_port%C5%AF_TCP_a_UDP>.

17. Resetovací útoky na TCP spojení. *LUPA* [online]. 2007, [cit. 2010-02-10]. Dostupný z WWW: <http://www.lupa.cz/clanky/resetovaci-utoky-na-tcp-spojeni/>.

18. *Rozhraní UNIX Sockets* [online]. [cit. 2010-02-10]. Http://www.cs.vsb.cz/grygarek/PS/sockets.html. Dostupné z WWW: <http://www.cs.vsb.cz/grygarek/PS/sockets.html>.

19. Poznáváme C# a Microsoft.NET – 53. díl – Timer a asynchronní delegáti. *Živě* [online]. 2005, [cit. 2010-2-16]. Dostupný z WWW: <http://www.zive.cz/clanky/poznavame-c-a-microsoftnet--53-dil--timer-a-asynchronni-delegati/sc-3-a-128231/default.aspx>.

20. File Transfer Protocol In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, 23. 9. 2004, 5. 3. 2010 [cit. 2010-03-17]. Dostupné z WWW: <http://cs.wikipedia.org/wiki/File_Transfer_Protocol>.

21. Simple Mail Transfer Protocol. *Http://www.cs.vsb.cz/grygarek/kotasek/smt01.htm* [online]. [cit. 2010-03-17]. Dostupný z WWW: <http://www.cs.vsb.cz/grygarek/kotasek/smt01.htm>.

22. POHOŘELÝ, Radovan. *IMAP (Internet Message Access Protocol)*. 2007. 12 s. Seminární práce. Dostupné z WWW: <http://st.vse.cz/~XPOHR04/>.

23. *Rebex.net: .NET components* [online]. [cit. 2010-04-01]. .NET Framework Class Library . Dostupné z WWW: <http://www.rebex.net/help/>.

24. Referenční model ISO/OSI - Prezentační vrstva. *Pc-site.owebu.cz* [online]. [cit. 2010-04-01]. Dostupný z WWW: <http://pc-site.owebu.cz/?page=ISO-OSI-14>.