VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

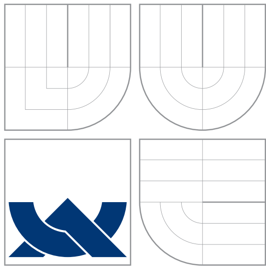
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

VYUŽITÍ PROTOKOLŮ TCP/IP PRO ŘÍZENÍ ALGORITMŮ ČÍSLICOVÉHO ZPRACOVÁNÍ ZVUKOVÝCH SIGNÁLŮ V REÁLNÉM ČASE

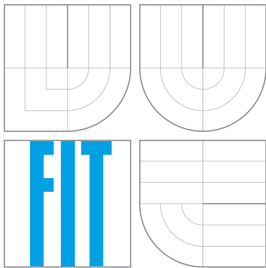
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE MARTIN PILCH
AUTHOR

BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

VYUŽITÍ PROTOKOLŮ TCP/IP PRO ŘÍZENÍ ALGORITMŮ ČÍSLICOVÉHO ZPRACOVÁNÍ ZVUKOVÝCH SIGNÁLŮ V REÁLNÉM ČASE

USE OF TCP/IP PROTOCOLS FOR ALGORITHM CONTROL IN REAL-TIME DIGITAL AUDIO SIGNAL PROCESSING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — MARTIN PILCH
AUTHOR

VEDOUCÍ PRÁCE — Doc. Dr. Ing. JAN ČERNOCKÝ
SUPERVISOR

BRNO 2009

Abstrakt

Tato bakalářská práce se zabývá návrhem protokolu pro řízení číslicového zpracování signálů pomocí komunikačních sítí TCP/IP. Využití tohoto protokolu v embedded jednotkách pro zpracování zvukového signálu umožní dálkové řízeného číslicového zpracování zvukových signálů v reálném čase. Knihovna, o které pojednává tato práce, umožňuje kvalitní a bezchybný přenos dat a tedy spolehlivé řízení zpracování zvukových signálů na velké vzdálenosti po lokální síti či internetu. Je však koncipovaná pro přenos libovolného typu dat a tak může nalézt i jiné uplatnění.

Abstract

This thesis deals with design of protocol for control of digital audio signal processing over TCP/IP networks. The application of this protocol in embedded audio processing units allows to remote control of real-time processing of digital audio signals. The library, this thesis disserts on, allows for superior and faultless transfer of data and accordingly reliable control of audio signal processing at long distances via the local network or Internet. This library is designed to transfer any type of data and can be applied in any sphere.

Klíčová slova

knihovna pro řízení datového přenosu, TCP/IP, UDP, C++, síťový protokol, zpracování zvukových signálů v reálném čase, knihovna WinSock, broadcast, multicast

Keywords

library for data transfer control, TCP/IP, UDP, C++, network protocol, real-time audio signal processing, WinSock library, broadcast, multicast

Citace

Martin Pilch: Využití protokolů TCP/IP pro řízení
algoritmů číslicového zpracování zvukových signálů v reálném čase, bakalářská práce, Brno, FIT VUT v Brně, 2009

Využití protokolů TCP/IP pro řízení algoritmů číslicového zpracování zvukových signálů v reálném čase

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jiřího Schimmela a pana Doc. Dr. Ing. Jana Černockého. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

....
Martin Pilch
18. května 2009

Poděkování

Děkuji Ing. Jiřímu Schimmelovi za konzultace a odborné vedení, stejně tak i panu Doc. Dr. Ing. Janu Černockému za pedagogické vedení.

© Martin Pilch, 2009.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Řízení parametrů systémů zpracování zvukových signálů 5

2 Síťové programování 7
2.1 Knihovna WinSock . . . 7
2.2 Historie knihovny WinSock . . . 7
2.3 Specifikace knihovny WinSock2 . . . 8
2.3.1 Socket . . . 8
2.3.2 Socket ve WinSock2 . . . 9
2.3.3 Vstupně - Výstupní rozhraní Overlapped I/O . . . 9
2.3.4 Události . . . 9
2.3.5 Vyhodnocení událostí . . . 10
2.3.6 Komunikace při vytvoření a ukončení spojení . . . 10
2.3.7 Přenos dat typu Broadcast . . . 11
2.3.8 Přenos dat typu Multicast . . . 11
2.4 Programování pod Windows . . . 12
2.4.1 Kritická sekce . . . 12
2.4.2 Vícevláknová aplikace . . . 13
2.4.3 Koncept I/O Completion Ports . . . 14

3 Etapy realizace 15
3.1 Komunikace pomocí blokujících socketů . . . 15
3.1.1 Návrh řešení problému . . . 15
3.1.2 Zhodnocení, problémy a nedostatky řešení . . . 16
3.2 Komunikace pomocí neblokujících socketů . . . 17
3.2.1 Návrh řešení . . . 17
3.2.2 Zhodnocení, problémy a nedostatky řešení . . . 18
3.3 Vícevláknová aplikace - nové vlákno pro každé přijaté spojení . . . 19
3.3.1 Návrh řešení . . . 19
3.3.2 Zhodnocení, problémy a nedostatky řešení . . . 20
3.4 Vícevláknová aplikace - řízení několika vláken pomocí IOCP . . . 21
3.4.1 Návrh řešení . . . 21
3.4.2 Zhodnocení, problémy a nedostatky řešení . . . 22
3.5 Přenos pomocí jednotlivých protokolů . . . 22
3.5.1 TCP . . . 23
3.5.2 UDP . . . 24
3.5.3 UDP broadcast . . . 24
3.5.4 UDP multicast . . . 25
3.5.5 Shrnutí . . . 26

3.6 Přenos dlouhých zpráv . . . 26

4 Důležité části knihovny **27**
4.1 Třída SocketInfo . . . 27
4.2 Třída ClientConnectionContext . . . 28
4.3 Třída ServerConnectionContext . . . 28
4.4 Pracovní vlákna a funkce pro přenos dat . . . 29
4.5 Třída dMatrixxClientClass . . . 29
4.6 Třída dMatrixxServerClass . . . 30
4.7 Důležité globální proměnné a konstanty . . . 30
4.7.1 Konstanty . . . 30
4.7.2 Globální proměnné . . . 31
4.7.3 Výčtový typ . . . 31

5 Použití knihovny **32**
5.1 Klientská část . . . 32
5.2 Serverová část . . . 33

6 Závěr **36**

A Přiložené CD **38**

Seznam obrázků

2.1 Architektura knihovny WinSock2. Přejato z webu [8, MSDN]. 8
2.2 Rozdíl mezi multicastem a broadcastem. 11
2.3 Schéma použití kritické sekce 13

3.1 Diagram serveru přijímajícího spojení pomocí blokujících socketů. 16
3.2 Diagram serveru přijímajícího spojení pomocí neblokujících socketů. 18
3.3 Diagram serveru vytvářejícího pro každého přijatého klienta nové vlákno. . 20
3.4 Diagram klienta pracujícího pomocí několika vláken řízených IOCP. 22

5.1 Diagram použití klienta. 32
5.2 Diagram použití serveru. 34

Úvod

Cílem této bakalářské práce je vytvoření knihovny pro řízení systémů zpracování zvukových signálů v reálném čase pomocí protokolů TCP/IP. Tato knihovna musí být schopna zajistit spolehlivý přenos dat a tím zajistit vzdálené ovládání těchto systémů.

Veškerá data pro řízení zpracování zvukových signálů je možné přenášet po síti a tím vzdáleně řídit zpracování zvukových signálů. V dnešní době je každý počítač vybaven rozhraním Ethernet a tak je možné řídit zpracování z jakéhokoliv počítače připojeného k síti nebo internetu.

Knihovna musí být tedy schopna pracovat ve dvou režimech. Buď jako klient a zvládnout například navázat spojení na několika aplikačních portech, každý s jiným protokolem a typem přenosu, či se v rámci každého aplikačního portu připojit k několika serverům, což obnáší mít pro každý aplikační port separátní seznam serverů. Nebo jako server a umožnit komunikaci na libovolném počtu aplikačních portů, každý s jiným protokolem a typem přenosu (unicast, multicast, broadcast) nebo například umožnit omezení počtu připojených klientů a filtrování podle seznamu povolených klientů. To obnáší vytvoření několika pracovních vláken a jejich řízení, pro přenos jednotlivých druhů informací jednotlivými vlákny. Pro spolehlivý přenos dat musí být ošetřeny veškeré chybové stavy a zajištěna bezeztrátovost přenosu.

Knihovna musí být také snadno použitelná a konfigurovatelná.

V první kapitole je zmíněno několik technologií pro řízení zpracování zvukových signálů.

V druhé kapitole jsou probrány teoretické předpoklady pro tuto práci týkající se síťového programování v prostředí Windows a vlastností nesouvisejících se síťovým programováním ale podstatných pro vypracování této práce.

Ve třetí kapitole jsou rozebrány jednotlivé etapy vývoje, jejich přínos a omezení.

Ve čtvrté jsou popsány jednotlivé části knihovny.

V páté a poslední kapitole je popsáno použití knihovny.

Kapitola 1

Řízení parametrů systémů zpracování zvukových signálů

Počátky standardizovaných rozhraní pro řízení systémů zpracování zvukových signálů je spojena s analogovými elektronickými hudebními nástroji. Nejpoužívanějším byl standard *CV/Trig*. S příchodem číslicově řízených systémů se začaly uplatňovat nové standardy číslicových rozhraní, jako je rozhraní [9, MIDI] (Music Instrument Digital Interface) pro řízení elektronických hudebních nástrojů nebo rozhraní *DMX 512* pro řízení světelné techniky. S rozvojem studiové techniky se začaly objevovat také protokoly pro řízení komplexních zvukových systémů ve vysílacích studiích. V současné době je v této oblasti nejrozšířenějším standardem rozhraní [6, EZ Bus].

Komplexnost moderních instalací multimediální systémů si vyžádala sjednocení rozhraní pro řízení zvukové, hudební, světelné, obrazové, scénické a multimediální techniky. Existuje řada firemních řešení, v oblasti standardizovaných rozhraní se o sjednocení snaží rozšíření protokolu MIDI jako je *MIDI Show Control*, *MIDI Machine Control* apod. nebo výše zmíněný protokol EZ Bus.

Posledním trendem ve vývoji rozhraní pro řízení multimediálních systémů je využití sítí Ethernet a IEEE 1394 jako nižších vrstev protokolů jiných rozhraní, jako je protokol [1, AM824], rozhraní EtherMIDI, Cobranet atd. Vznikají také nová řešení na míru těchto rozhraní, např. [2, AV/C Command Language], [13, mLAN] a další. Pro sledování činnosti např. rádiových a televizních vysílacích systémů se začíná využívat protokolu [7, SNMP] (Simple Network Management Protocol), který byl původně navržen pouze pro správu aktivních síťových prvků.

V současné době tvoří většinu systémů pro zpracování zvukových signálů počítačové nebo počítačově řízené systémy. Algoritmy pro zpracování signálů jsou realizovány formou tzv. *plug-in modulů*, které jsou jako samostatné procesy zaváděny a spouštěny v rámci řídicí aplikace buďto přímo v centrální procesorové jednotce počítače nebo v signálových procesorech na přídavných kartách instalovaných v rámci počítačového systému. Aby byla zajištěna univerzálnost a přenositelnost plug-in modulu pro zpracování signálů, tj. aby bylo možné jej využít v různých hostitelských aplikacích, je nutné vytvořit a zveřejnit standardizované aplikační rozhraní pro přenos zpracovávaných dat a komunikaci mezi plug-in modulem a hostitelskou aplikací, tzv. rozhraní plug-in modulu. V závislosti na konkrétní technologii je rozhraní plug-in modulu buďto součástí operačního systému či jeho rozšíření nebo je součástí samotné hostitelské aplikace. Rozšířenou technologií, která podporuje používání plug-in modulů, je např. technologie [4, Microsoft DirectX].

Rozhraní pro řízení parametrů algoritmů číslicového zpracování signálu musí být také standardizováno, aby bylo možné současné řízení parametrů různými prostředky, např. pomocí grafického uživatelského rozhraní, automatizace dat v hostitelské aplikaci, externími ovládacími pulty atd. Požadavek univerzálnosti řídicího rozhraní pro různé typy algoritmů a jejich různé parametry vyžaduje tzv. mapování parametrů algoritmu do interních parametrů řídicího protokolu. Tímto způsobem pracují moderní technologie používané pro implementaci algoritmů číslicového zpracování zvukových signálů *VST*, *Audio Units*, *RTAS* a další. Parametr takového rozhraní je datovým blokem, jehož interpretaci musí zajistit samotný plug-in modul podle pravidel řídicího rozhraní daného systému [5].

Využití protokolů sady TCP/IP je výhodné z hlediska využití stávající infrastruktury, komunikačních komponentů (každý osobní počítač je vybaven rozhraním Ethernet a operační systém disponuje subsystémem pro síťovou komunikaci), spojové a datagramové služby, způsobu adresování cílových stanic a jejich sdružování atd.

Moderní systémy pro zpracování zvukových signálů v reálném čase musí kromě řídicích parametrů přenášet také kontrolní a konfigurační data a binární soubory (např. pro dálkové zavedení nových plug-in modulů do systému). Na tyto přenosy jsou kladeny různé nároky, některé musí být přenášeny pomocí spolehlivé služby, jiné naopak co nejrychleji. Požadavkem je také možnost řízení systému z několika ovládacích stanic současně a naopak monitoring více systémů z jedné kontrolní aplikace. Mezi požadavky na systém vytvořený v rámci této práce je proto možnost paralelní komunikace na několika aplikačních portech současně pomocí různých protokolů, realizace serveru a vícenásobného klienta.

Kapitola 2

Síťové programování

V této kapitole budou probrány některé aspekty programování síťových aplikací a aplikací pod operačními systémy Windows obecně.

2.1 Knihovna WinSock

Windows Sockets API, zkráceně *WinSock*, je technická specifikace, definující jak by měly síťové aplikace naprogramované pod operačním systémem Microsoft Windows přistupovat k síťovým službám, nejčastěji TCP/IP. Názvosloví je odvozeno od *Berkeley sockets API* nebo také *BSD Sockets* modelu používaného v operačních systémech založených na *Berkeley UNIX*.

Jako zdroje byl použit MSDN web [8, http://www.msdn.com], kniha Josefa Pirkla [11, Síťové programování pod Windows a programování Internetu] a kniha [10, Illustrated TCP/IP, A Graphic Guide to the Protocol Suite].

2.2 Historie knihovny WinSock

První operační systémy firmy Microsoft (MS-DOS a Microsoft Windows) disponovaly omezenou schopností síťové komunikace, založené hlavně na protokolech NetBIOS/NetBEUI, které ve starších verzích nebyly schopné například směrování.

Mnoho univerzitních skupin i komerčních vývojářů jako Sun Microsystems nebo Distinct tedy vyvinulo produkty pro MS-DOS podporující protokol TCP/IP. Všichni však používali vlastní aplikační rozhraní (Application Programing Interface, API) a chyběl standard, který by dovolil komukoliv vytvořit si vlastní síťovou aplikaci, aniž by musel používat jednu z implementací.

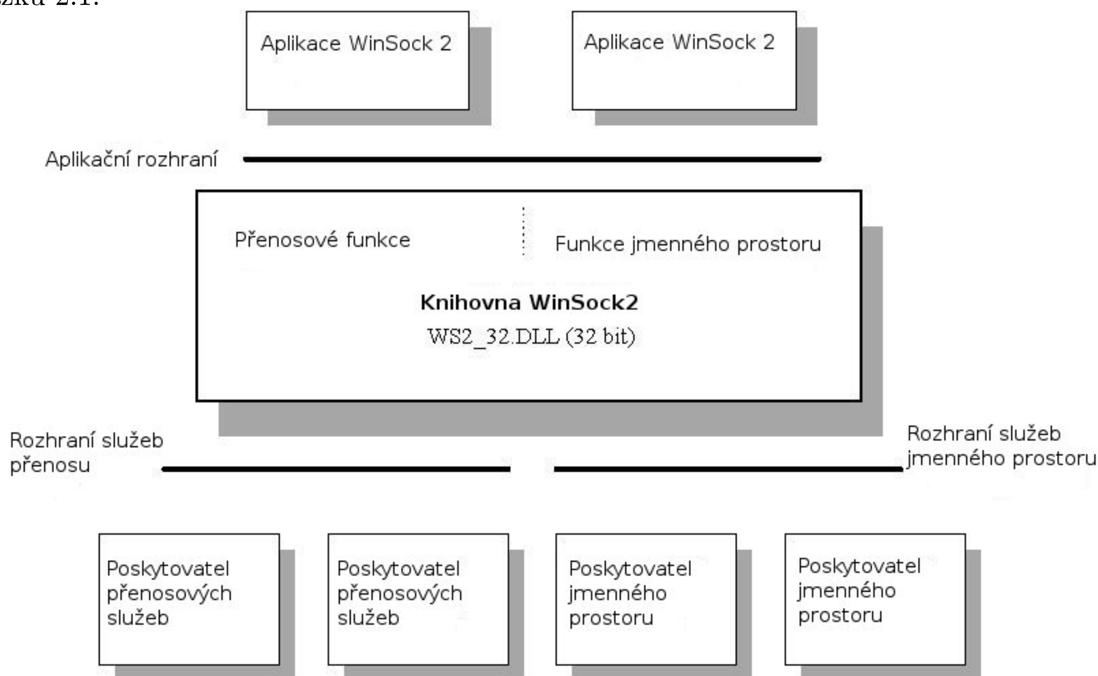
První verzi *Windows Sockets API* navrhli Martin Hall, Mark Towfiq, Geoff Arnold, Henry Snaders a J. Allard za asistence mnoha dalších.

Samotný Microsoft zpočátku knihovnu WinSock nepodporoval. Do svých operačních systémů zařadil až verzi 1.1.

Postupně se knihovna vyvíjela. Byly přidávány nové funkce jako například podpora multicastingu, vyhodnocování kvality přenosu, asynchronních operací a událostí. Nebo také sdílení socketů mezi procesy, provádění operací nad skupinou socketů či podmíněné akceptování spojení.

2.3 Specifikace knihovny WinSock2

Specifikace Windows Sockets API definuje dvě rozhraní. Prvním je *API* (aplikační rozhraní), používané vývojáři aplikací, druhým pak *SPI* (rozhraní služeb), které poskytuje prostředky pro přidání nových modulů protokolů do systému (specifikace nových protokolů). *Aplikační rozhraní* zaručuje, že korektně napsaná aplikace bude fungovat se správně naprogramovanou implementací protokolu. Obdobně *rozhraní služeb* zaručuje, že koreketně napsaný modul bude v operačním systému bezproblémově fungovat se správně napsanou aplikací. Tímto byla zajištěna možnost naprogramovat korektně pracující síťovou aplikaci. Vše je vidět na obrázku 2.1.



Obrázek 2.1: Architektura knihovny WinSock2. Přejato z webu [8, MSDN].

Knihovna Windows Sockets je založena na BSD socketech ale sjednocuje programátorské rozhraní s tím z Windows. Některé funkcionality BSD socketů nemohly být použity v knihovně WinSock2 kvůli rozdílnosti mezi Windows a UNIXem. Na druhou stranu byly jiné funkce rozšířeny, například podpora neblokujících či asynchronních socketů.

Podrobněji je specifikace rozebrána v následujících podkapitolách. Budou zmíněny pouze vlastnosti podstatné pro tuto bakalářskou práci. Kompletní specifikace se nachází na webu MSDN `http://msdn.microsoft.com/en-us/library/ms740673(VS.85).aspx`.

2.3.1 Socket

Socket je koncový bod dvousměrného datového přenosu po síti, založené na protokolu *IP* (Internet Protocol). Typicky je touto sítí lokální počítačová síť nebo internet.

Socket je charakterizován unikátní kombinací *protokolu* (TCP, UDP nebo raw IP), *lokální adresy socketu* (IP adresy počítače a čísla portu) a u socketů protokolu TCP, tedy spojovaných i *adresou vzdáleného socketu.*

Pro operační systém a aplikaci, která jej vytvořila, je socket reprezentován unikátním 32-bitovým celým číslem, *identifikátorem socketu.*

2.3.2 Socket ve WinSock2

V knihovně WinSock2 je socket reprezentován jako ukazatel na objekt. Je s ním možno pracovat jako se souborem pomocí standartních funkcí operačního systému Windows `ReadFile(), WriteFile(), ReadFileEx() a WriteFileEx()`. Lepší je však použít funkce nabízené přímo knihovnou WinSock2, a to `WSARecv(), WSASend() a WSADuplicateSocket()`, které umožňují např. nastavení událostí nez zásahu programátora. Dalším důvodem je možnost lepšího ošetření chyb vzniklých během použití funkcí knihovny WinSock2 pomocí funkce `WSAGetLastError()`, narozdíl od výše zmíněných standartních, které generují systémové chyby a neinformují o přesné příčině.

Je důležité vysvětlit rozdíl mezi blokujícími a neblokujícími sockety. *Blokující* jsou takové, které čekají na dokončení operace. Například program očekává data a nepokračuje v provádění kódu, dokud je neobdrží.

Neblokující naopak toto zablokování nezpůsobí. Program pouze čeká na událost oznamující například přicházející data a až po výskytu této události se provede příslušná operace.

Jako nejefektivnější se jeví použití neblokujících socketů, případně kombinace obou přístupů.

2.3.3 Vstupně - Výstupní rozhraní Overlapped I/O

Overlapping neboli překrývání se používá pro odesílání či příjem dat, při kterém se operační systém postará o co nejvyšší výkon přenosu.

Při odesílání, ať už funkcí `WSASend()` nebo `WSASendTo()`, jsou data nakopírována do vyrovnávací mezipaměti a o odeslání se postará operační systém. Obdobně při příjmu jsou data zkopírována z přenosové mezipaměti do vyrovnávací mezipaměti a čekají na vyzvednutí pomocí funkce `WSARecv()` nebo `WSARecvFrom()`.

Tyto funkce okamžitě vrací návratovou hodnotu. Při okamžitém odeslání nebo příjmu dat je to nula. Při chybě vrací hodnotu `SOCKET_ERROR`. Pokud se jedná o chybu `WSA_IO_PENDING`, operace odeslání nebo příjmu stále probíhá a je potřeba počkat na její dokončení. V opačném případě nastala chyba při přenosu.

Nevýhodou při tomto způsobu přenosu je riziko, že zprávy nedorazí v pořadí, v jakém byly odeslány, respektive, jejich příjem nemusí být indikován ve správném pořadí. Je proto potřeba do zpráv vložit hlavičku s pořadím a programově zajistit jejich zpětné složení.

Pro overlapped přenos je možné použít pouze sockety vytvořené funkcí `WSASocket()` s atributem `WSA_FLAG_OVERLAPPED` a jako jeden z parametrů funkcí pro příjem a odeslání je potřeba uvést `WSAOVERLAPPED` strukturu, která obsahuje informace pro overlapped komunikaci (například událost, která je nastavena při dokončení přenosu).

2.3.4 Události

K problematice síťového programování pomocí WinSock se úzce váže problematika událostí. Ať už se jedná o neblokující sockety, kde událost oznamuje žádost klienta o spojení či zaslání dat, nebo o rozhraní overlapped, kde událost indikuje dokončení přenosu.

Pro práci s událostmi se používají tyto funkce:

- `WSACreateEvent()` pro vytvoření nového objektu události.
- `WSACloseEvent()` pro uzavření objektu události.
- `WSASetEvent()` pro nastavení objektu události, indikaci toho, že událost nastala.
- `WSAResetEvent()` pro vyresetování objektu události. Používá se většinou po reakci na událost.
- `WSAEventSelect()` pro přiřazení síťových událostí FD_XXX socketu, na kterém bude očekávána a události, která bude nastavena po vyvolání jedné z těchto událostí.

2.3.5 Vyhodnocení událostí

Programátor tak má možnost, jak vyhodnotit a příslušně reagovat na události, které nastaly.
Existují tři způsoby, jak je vyhodnotit:

- Čekání na indikaci jedné či více událostí pomocí funkce `WSAWaitForMultipleEvents()`. Proces nebo vlákno je během čekání zablokováno. Pokud aplikace využívá rozhraní overlapped, je potřeba ještě vyhodnotit provedenou operaci následující funkcí.
- Dotazování na dokončení overlapped operace funkcí `WSAGetOverlappedResult()`. Pomocí této funkce je možné zjistit, jestli byla overlapped operace dokončena či ne. Touto funkcí je možné také čekat na provedení overlapped operace a tím proces nebo vlákno zablokovat
- Procedura zadaná jako nepovinný parametr funkcí `WSASend()`, `WSASendTo()`, `WSARecv()` a `WSARecvFrom()`, která je zavolána po dokončení overlapped operace, ať už úspěšném či neúspěšném.

Pro vyhodnocení overlapped operací (pokud je v aplikaci použito overlapped rozhraní), je možno použít jakýkoliv z těchto tří způsobů vyhodnocení událostí. V opačném případě, například pro čekání na připojení klienta, pouze první způsob.

2.3.6 Komunikace při vytvoření a ukončení spojení

Pomocí parametru funkce `WSAConnect()` lze při připojování klienta k serveru zaslat druhé straně jakákoliv data. Na programátorovi pak je, aby při programování serveru tato data nějakým způsobem využil po přijetí funkcí `WSAAccept()`.

Podobným způsobem lze před ukončením spojení zaslat data funkcí `WSASendDisconnect()` a na druhé straně (tedy serveru) po indikaci události `FD_CLOSE` (například ve funkci `recv()` či `WSARecv()`) přijmout funkcí `WSARecvDisconnect`, vyvolat `WSASendDisconnect`, čímž se klientovi odešle indikace `FD_CLOSE`, kterému už stačí provést `WSARecvDisconnect()` a uzavřít socket (to je potřeba udělat i na straně serveru).

Tímto způsobem je možné komunikovat před vlastním připojením a odpojením od druhou stranou. Je možné například klienta aktivně odmítnout.

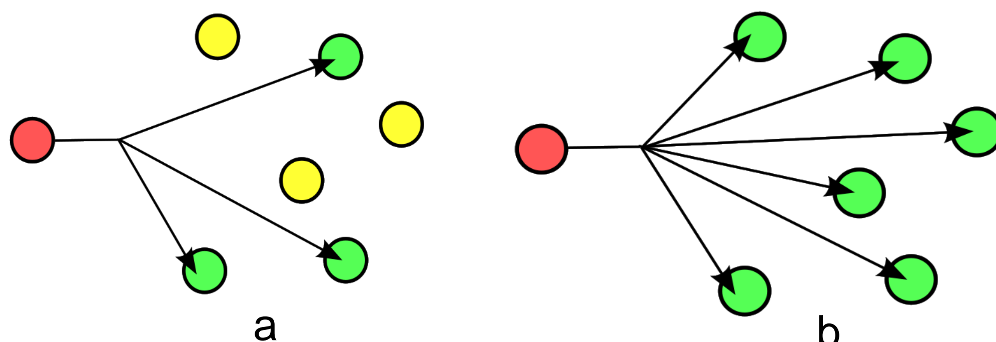
2.3.7 Přenos dat typu Broadcast

Broadcastem se rozumí packet, který je odeslán všem počítačům v síti. Výhoda spočívá v tom, že se dá jednou operací obsloužit velký počet žadatelů (samozřejmě pokud je jim potřeba odeslat stejná data), nevýhodou je způsobené velké vytížení sítě, kdy se paket posílá i těm počítačům, které ho nevyžadují.

Broadcast využívá pro svou komunikaci protokolu *UDP*, není jej možné použít se spojovanými protokoly (například TCP). Pro použití je potřeba upravit nastavení socketu pomocí funkce `setsockopt()` povolením `SO_BROADCAST` volby. Poté už jen stačí odeslat packet na broadcastovou adresu. Univerzální adresa je *255.255.255.255*, kterou však některé routery filtrují. Rozumnější volbou je vypočítání broadcastové adresy a to logickým součinem masky sítě s IP adresou počítače, ze kterého se packet posílá. Tuto hodnotu (adresu sítě) je potřeba logicky sečíst s negovanou hodnotou masky sítě. Tím získám broadcastovou adresu, kterou nebudou routery filtrovat, a broadcastové packety si najdou cestu k cíli.

2.3.8 Přenos dat typu Multicast

Zásadní výhoda multicastového adresování oproti broadcastu je v menším vytížení síťe. Packet se odesílá pouze těm příjemcům, kteří jsou zaregistrováni v multicastové skupině. Navíc se neposílá každému zvlášť, ale pošle se pouze jednou a na routeru, ke kterému jsou připojeny zaregistrovaní přijemci, se teprve rozdělí a odešlou každému ze zaregistrovaných. Názorně to ilustruje obrázek 2.2, kde je znázorněno multicastové (a) a broadcastové adresování (b):



Obrázek 2.2: Rozdíl mezi multicastem a broadcastem.

Multicast, stejně jako broadcast, pracuje s nespojovanými protokoly (tedy především UDP). Opět se používá funkce `setsockopt()` pro nastavení parametrů socketu. Nejdůležitějšími parametry jsou `IP_ADD_MEMBERSHIP` a `IP_DROP_MEMBERSHIP`. Prvním se přihlásí počítač do multicastové skupiny pokud chce data přijímat, druhým se členství ve skupině vzdá. Jedním z parametrů pak musí být struktura *in_addr* obsahující lokální IP adresu a adresu multicastové skupiny. Je možné taky použít paramter `IP_MULTICAST_LOOP`, kterým lze nastavit, zda se odeslaná data budou vracet k odesílateli.

Dalším důležitým parametrem je `IP_MULTICAST_TTL` (Time-To-Live), kterým lze nastavit, jak daleko se dostanou multicastové packety, než budou jako nedoručené zničeny. Tento parametr lze nastavit na tyto hodnoty:

- 0 - omezení na aktuální počítač

- 1 - omezení na aktuální podsíť
- 32 - omezení na aktuální síť
- 64 - omezení na aktuální region
- 128 - omezení na aktuální kontinent
- 255 - bez omezení

Tyto hodnoty jsou převzaty z webu MSDN [8, http://www.msdn.com] a jsou pouze orientační. Pro účely této práce se jeví jako nejvhodnější omezení na aktuální síť.

2.4 Programování pod Windows

V této sekci bude probráno několik vlastností nabízených operačním systémem nesouvisejících s knihovnou WinSock, ale které unadnily vypracování této práce. Jedná se především o multithreading, tedy o tvorbu vícevláknových aplikací, a IOCP (Input/Output Completion Port), usnadňující jejich řízení.

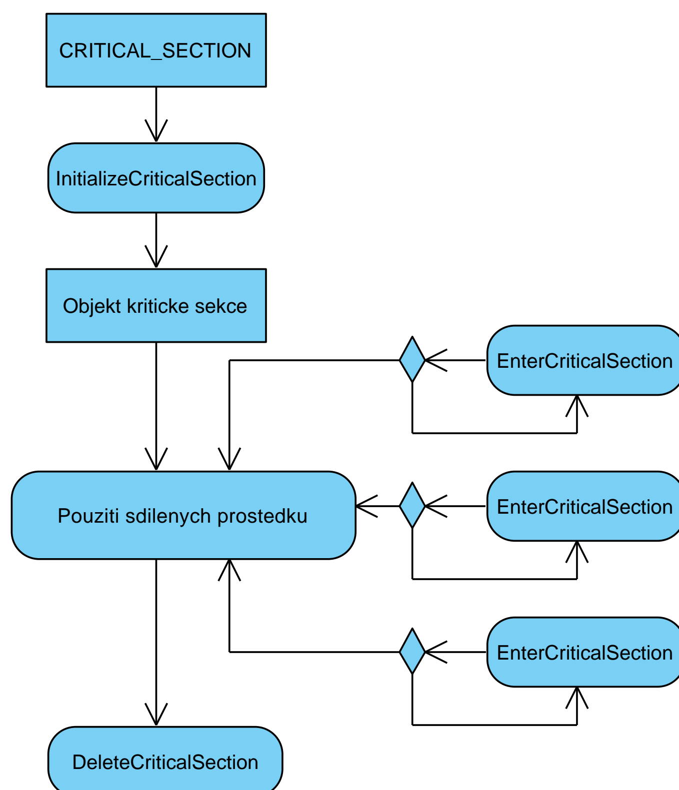
2.4.1 Kritická sekce

Kritická sekce reprezentuje objekt, pomocí něhož je programátor schopný sdílet prostředky mezi programovými vlákny. Princip spočívá v tom, že objekt kritické sekce může být zabrán pouze jedním vláknem a ostatní čekají na jeho uvolnění. Takto je možný přístup ke sdíleným prostředkům bez konfliktu, jakým je například změna obsahu vektoru dvěma vlákny zároveň.

Proces je zodpovědný za přidělení a uvolnění paměti pro objekt kritické sekce. Paměť se přidělí prostým deklarováním proměnné typu *CRITICAL_SECTION* a inicializací objektu funkcí `InitializeCriticalSection()`. Po skončení práce s objektem je potřeba prostředky uvolnit pomocí funkce `DeleteCriticalSection()`. Objekt pak už není možné použít pro sdílení prostředků.

Pro zabrání objektu kritické sekce a tedy možnosti přistupovat ke sdíleným prostředkům se používá funkce `EnterCriticalSection()` nebo `TryEnterCriticalSection()`. Pro uvolnění objektu pak `LeaveCriticalSection()`. Pokud je kritická sekce zabrána jiným vláknem, vlákno pokoušející se o přístup k tomuto objetku čeká na uvolnění v již zmíněné funkci `EnterCriticalSection()`. Takto je vytvořena fronta typu *FIFO* (First-In-First-Out) ze všech čekajících vláken. Použitím funkce `TryEnterCriticalSection()` lze zabránit čekání vlákna ve frontě. Tato funkce totiž vrací okamžitě výsledek. Pokud je objekt volný, vlákno nad ním získá kontrolu, pokud ne, je vrácena nulová hodnota a vlákno tak nemusí čekat ve frontě a může případně vykonávat jinou činnost a pokusit se o přístup později.

Na obrázku 2.3 lze vidět schéma použití kritické sekce a ve zkratce zobrazuje to, co jsem popisoval výše. Po deklaraci objektu a jeho inicializaci je kritická sekce připravena k použití. Vlákna se pokoušejí o přístup k objektu a buď jim je povolen (jednomu z nich) nebo čekají na uvolnění. Nakonec jsou alokované prostředky uvolněny.



Obrázek 2.3: Schéma použití kritické sekce

Kritická sekce je mocný nástroj pro sdílení prostředků mezi více vlákny. Není ji však možné použít v aplikaci s více procesy. V tomto případě se používají objekty typu *Vzájemného vyloučení* (Mutex), *Semafor* (Semaphore) nebo *Události* (Event). Specifikace se nachází na webu MSDN `http://msdn.microsoft.com/en-us/library/ms682530.aspx`.

2.4.2 Vícevláknová aplikace

Aplikace jako taková se skládá z jednoho či více *procesů*. Proces můžeme chápat jako program. *Vlákno*, anglicky *thread*, je základní jednotkou, které operační systém přiděluje *procesorový čas*. V procesu může být spuštěno jedno či více vláken, přičemž vlákno může vykonávat jakoukoliv část kódu procesu. Nejdůležitější vlastností je, že více vláken může vykonávat stejnou část procesu. Sdílejí kontext procesu. Více například na webu MSDN `http://msdn.microsoft.com/en-us/library/ms684841(VS.85).aspx`.

S tím úzce souvisí pojem *Multitasking*, česky přepínání mezi vlákny. Operační systém podporující multitasking rozděluje dostupný procesorový čas mezi potřebné procesy a vlákna.

Každému vláknu (či procesu) je přiřazena část procesorového času. V momentě, kdy ji spotřebuje, je jeho vykonávní přerušeno, uložen jeho kontext (data, místo, kde skončil s vykonáváním atd.) a z fronty se vyjme další vlákno, jehož kód se následně vykonává po přidělený procesorový čas.

Protože přidělená část procesorového času je velmi malá (desítky milisekund), operační systém zdánlivě provádí více operací najednou (u vícejádrových procesorů pracuje zároveň

tolik vláken, kolik má procesor jader, tedy nejen zdánlivě), například rozbalování archívu, zatímco uživatel upravuje textový soubor.

Každý proces obsahuje prostředky pro svůj běh. Jsou to jedinečný identifikátor procesu, virtuální adresový prostor, kód, který proces provádí a, mimo jiné, i minimálně jedno takzvané *primární vlákno*. Při spuštění procesu je toto vlákno vytvořeno, proces pak může vytvářet další vlákna.

Vlákna procesu sdílí virtuální adresový prostor a systémové prostředky. Dále pak každé vlákno obsahuje svůj vlastní jedinečný identifikátor, paměťový prostor a další prostředky. především však obsahuje struktury pro uložení kontextu během pozastavení vykonávání kódu vlákna při multitaskingu.

Využití naleznou vícevláknové aplikace všude tam, kde je potřeba provádět relativně rutiní operaci a zároveň pokračovat v další práci. Proces, který vytvořil pracovní vlákna, totiž nečeká na jejich dokončení. Může tedy vláknům, starajícím se například o odeslání dat, předat data k odeslání a provádět cokoliv jiného. Synchronizace mezi vlákny pak probíhá pomcí již zmíněných událostí, kritické sekce nebo I/O Completion portů, které popisuje následující kapitola.

2.4.3 Koncept I/O Completion Ports

Pro řízení vícevláknové aplikace je vhodné použít koncept *IOCP* (Input/Output Completion Ports). Vytvořením I/O completion portu systém vytvoří frontu, pomocí níž lze řídit požadavky rychleji a efektivněji. Informace byly čerpány z příslušné sekce MSDN http://msdn.microsoft.com/en-us/library/aa365198(VS.85).aspx.

Pomocí funkce `CreateIoCompletionPort()` se vytvoří I/O completion port a zároveň se spojí například se socketem či mail slotem, který se zadá jako jeden z parametrů této funkce. V mém případě se jednalo o socket. Takto je možné jednomu portu přiřadit i více socketů. V momentě, kdy je na jednom ze socketů dokončena asynchronní I/O operace, do fronty typu *FIFO* (firt-in-first-out) je přidán *I/O completion paket*, který, mimo jiné, obsahuje *CompletionKey*. Pomocí tohoto klíče je možno předat libovolný parametr a tím řídit běh aplikace (může jím být například adresa odkazující na řetězec pro odeslání). Důležitým parametrem této funkce je parametr *NumberOfConcurrentThreads*, udávající počet vláken, které budou zpracovávat frontu.

Funkcí `PostQueuedCompletionStatus()` je možné přidat completion paket do fronty a tím řídit běh aplikace. Je to jedna z možností, jak komunikovat mezi vlákny.

Po vytvoření completion portu se v každém vlákně čeká funkcí `GetQueuedCompletionStatus()` na completion paket, který se objeví ve frontě pro zpracování (dobu čekání lze samozřejmě ovlivnit jedním z parametrů). Jedno z čekajících vláken z fronty vyjme paket a tato funkce vrátí nenulovou hodnotu. Vlákno pak může použít completion klíč pro svou další práci. Je tak možno reagovat na I/O operaci nebo komunikaci z jiného vlákna.

Hlavním přínosem konceptu IOCP je efektivní řízení vícevláknové aplikace a komunikace mezi aplikací a vytvořenými vlákny.

Kapitola 3

Etapy realizace

V následujících sekcích budou rozebrány jednotlivé etapy vývoje. V každé z nich pak je zmíněno, co je jejím cílem, nastíněna možná řešení, rozebrány problémy vzniklé během implementace a navrženo řešení nedostatků.

Pokud jsou zmíněny funkce ať už knihovny WinSock či jiné, vždy je uveden pouze její název bez parametrů. Jejich deklarace pak je možné najít v drtivé většině případů na stránkách MSDN [8, http://www.msdn.com].

Jako programovací jazyk byl pro svou rychlost zvolen objektově orientovaný jazyk C++ [3, http://www.cplusplus.com/reference/].

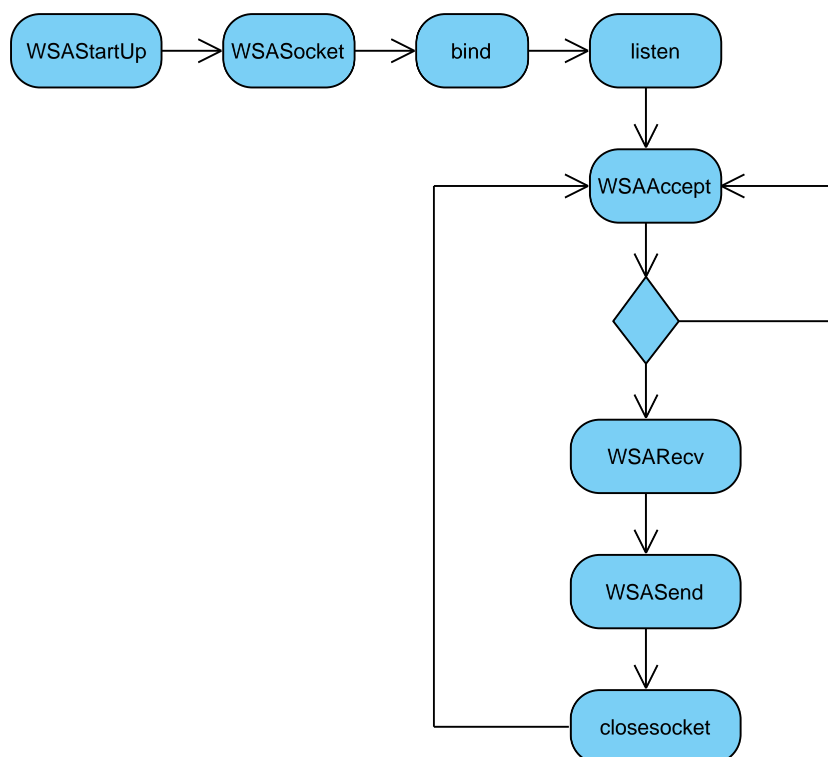
3.1 Komunikace pomocí blokujících socketů

Jako první krok při práci na této práci byla naprogramována aplikace, pracující s blokujícími sockety. Toto řešení je jednoduché na realizaci, má však jeden zásadní nedostatek, který jej odsuzuje k použití v jednoduchým aplikacích.

Program umí přijmout klientovu žádost o spojení, zprávu od klienta a vypsat ji na obrazovku. Analogicky se klient spojí se serverem a pošle zprávu zadanou uživatelem.

3.1.1 Návrh řešení problému

Pomocí diagramu na obrázku 3.1 je popsán návrh řešení serveru komunikujícího pomocí blokujících socketů.



Obrázek 3.1: Diagram serveru přijímajícího spojení pomocí blokujících socketů.

Ze všeho nejříve je potřeba inicializovat knihovnu WinSock funkcí `WSAStartup()` s parametrem, udávajícím verzi WinSock knihovny, která se má použít. V opačném případě samozřejmě nebude možno použít žádnou z funkcí, které nabízí.

Poté je potřeba vytvořit socket funkcí `WSASocket()`, na kterém bude aplikace naslouchat a pomocí něhož bude přijímat spojení.

Funkce `bind()` zajistí přiřazení portu vytvořenému socketu, určí, pomocí jakých protokolů se bude komunikovat (například `AF_INET`) a z jakých IP adres se bude přijímat spojení (běžné se používá `INADDR_ANY`).

Aby se dala na tomto socketu přijímat spojení, je ještě potřeba nastavit naslouchání a to funkcí `listen()`. Nyní už nic nebrání akceptování klientů.

`WSAAccept()` očekává příchozí spojení. V momentě, kdy se objeví, přijme požadavek klienta a čeká na příchozí data. Po přijetí dat pomocí `WSARecv()` je vypíše a odešle zpět funkcí `WSASend()`. Následně se spojení ukončí zavřením socketu `closesocket()` a opět se čeká na další spojení.

3.1.2 Zhodnocení, problémy a nedostatky řešení

Jak bylo zmíněno výše, způsob komunikace pomocí blokujících socketů je vhodný pouze pro použití v jednoduchých aplikacích a to z jednoho důvodu. Server při čekání na spojení od klienta funkcí `WSAAccept()` čeká, spotřebovává procesorový čas a nemůže vykonávat jinou

činnost. Díky tomu lze pomocí tohoto řešení obsloužit pouze jednoho klienta. Ostatní, pokoušející se o spojení, musí počkat. Tento problém lze vyřešit dvěma způsoby:

- Použitím neblokujících socketů, kdy server nejdříve funkcí `WSAEventSelect()` nastaví příslušnému socketu události, které na něm mohou nastat a poté kontroluje jejich výskyt. Až poté akceptuje spojení pomocí funkce `WSAAccept()`.
- Vytvořením jednoho či více vláken, ve kterých bude server akceptovat spojení. Činnost programu tak nebude zablokována, stále však bude aktivně čekat na připojení. Programátorsky se však jedná o náročnější řešení, je nutné vytvořit režii vláken.
- Kombinací obou předchozích způsobů lze vytvořit neblokující server, schopný kromě čekání na spojení vykonávat i jinou činnost.

Pokud je tedy potřeba efektivně obsloužit více klientů najednou, je nutné použít jedno z těchto řešení.

3.2 Komunikace pomocí neblokujících socketů

Pro složitější aplikace je vhodné použít neblokujících socketů, které nečekají na dokončení operace, a tudíž neblokují aplikaci. Místo toho neblokující sockety používají pro zjištění, jaká se bude provádět operace jeden z těchto dvou přístupů:

- Takzvaný *polling*, kdy se aplikace periodicky, nejčastěji pomocí časovače, dožaduje například příjmu dat.
- Čekání na síťovou událost oznamující požadavek na jednu z operací.

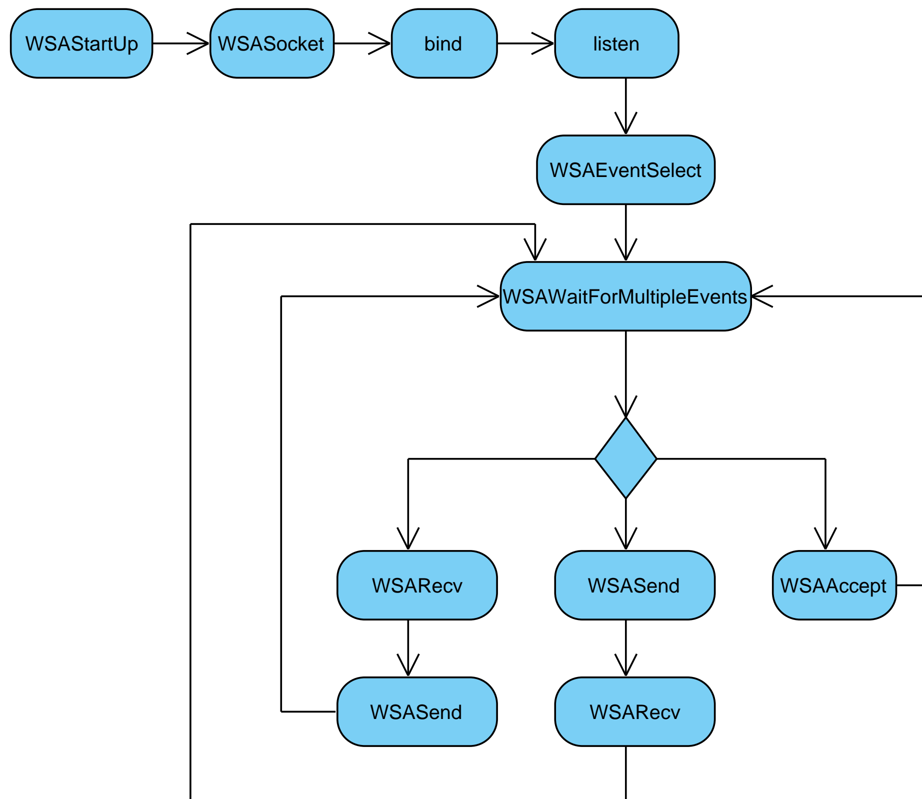
Výhodnější se jeví čekání na výskyt síťové události, než opakované provádění operace. Aplikace bude provádět stejnou činnost jako předchozí řešení. Server přijme spojení od klienta, poté data od něj, které mu pošle nazpět. Rozdíl je v použití neblokujících socketů a v tom, že aplikace po přijetí dat a jejich zpětném odeslání neukončí spojení s klientem.

3.2.1 Návrh řešení

Inicializace knihovny WinSock, vytvoření socketu pro naslouchání, přiřazení portu tomuto socketu a zajištění naslouchání je shodné s předchozím řešením. O určení, které síťové události budou na daném socketu signalizovány, se stará funkce `WSAEventSelect()`, inicializující programátorem určený *HANDLE* události. Čekání na jejich výskyt pak obstará funkce `WSAWaitForMultipleEvents()`. U serveru se čeká na jednu ze dvou událostí. Jedna z nich oznamuje žádost o spojení klienta připojujícího se k socketu, na kterém server naslouchá. Druhá pak žádost o přenos dat jedním, či druhým směrem na socketu vzniklém při akceptování spojení (komunikační socket). Na straně klienta se pak touto funkcí vyhodnocuje jediný *HANDLE* události, kterému je přiřazen socket a požadované síťové události, tedy `FD_CONNECT`, `FD_READ` a `FD_WRITE`.

Aplikace tedy čeká na výskyt síťové události `FD_ACCEPT` na straně serveru, oznamující žádost o spojení. Po přijetí spojení funkcí `WSAAccept()` se na straně klienta objeví síťová událost `FD_CONNECT` oznamující právě ono přijetí klientovy žádosti o spojení. Poté server čeká na výskyt události `FD_READ`, která se objeví při pokusu klienta o odeslání dat funkcí `WSASend()`, a na základě které bude číst data ze socketu pomocí funkce `WSARecv()`. Data

pošle zpět, klientovi se samozřejmě také objeví událost oznamující příchozí data a ten je přijme. Pokud by klient chtěl data od serveru přijímat (funkcí `WSARecv()`) a ne je odesílat, na straně serveru by se objevila událost `FD_WRITE`. Touto by byl požádán o zápis dat do socketu.



Obrázek 3.2: Diagram serveru přijímajícího spojení pomocí neblokujících socketů.

Na diagramu na obrázku 3.2 je vidět, že server dokáže, na rozdíl od předchozího řešení, obsloužit více klientů (nedokáže je obsloužit zároveň). Důležité je však čekat na výskyt události na naslouchacím socketu a komunikačních socketech připojených klientů.

3.2.2 Zhodnocení, problémy a nedostatky řešení

Použitím neblokujících socketů byl vytvořen server, který je schopen obsloužit více klientů, aniž by musel ukončit spojení s jiným klientem. Tím byl splněn druhý bod zadání. Neblokující sockety představují, proti blokujícím socketům, mocný nástroj při vývoji síťových aplikací. Stále však nemůže obsluhovat více klientů najednou. Toho je možné dosáhnout pomocí multithreadingu, tedy vícevláknové aplikace. Nabízí se dvě možnosti řešení:

- Vytvoření nového pracovního vlákna pro každého přijatého klienta.

- Vytvoření několika pracovních vláken a jejich řízení pomocí konceptu *IOCP*.

V následujících podkapitolách budou popsána obě řešení a zhodnoceny jejich výhody a nevýhody.

3.3 Vícevláknová aplikace - nové vlákno pro každé přijaté spojení

Jak již bylo zmíněno výše, obsloužit více klientů najednou je možné pomocí vícevláknové aplikace. Tato souběžná obsluha je u jednojádrových procesorů pouze zdánlivá, protože však dochází k přepínání mezi vlákny ve velmi kráktých intervalech, jeví se lidskému oku nepřerušovaně. U vícejádrových procesorů je procesor schopen zpracovat tolik vláken najednou, kolik má jader.

Koncepce jednoho vlákna pro každé spojení byla použita pro stranu serveru.

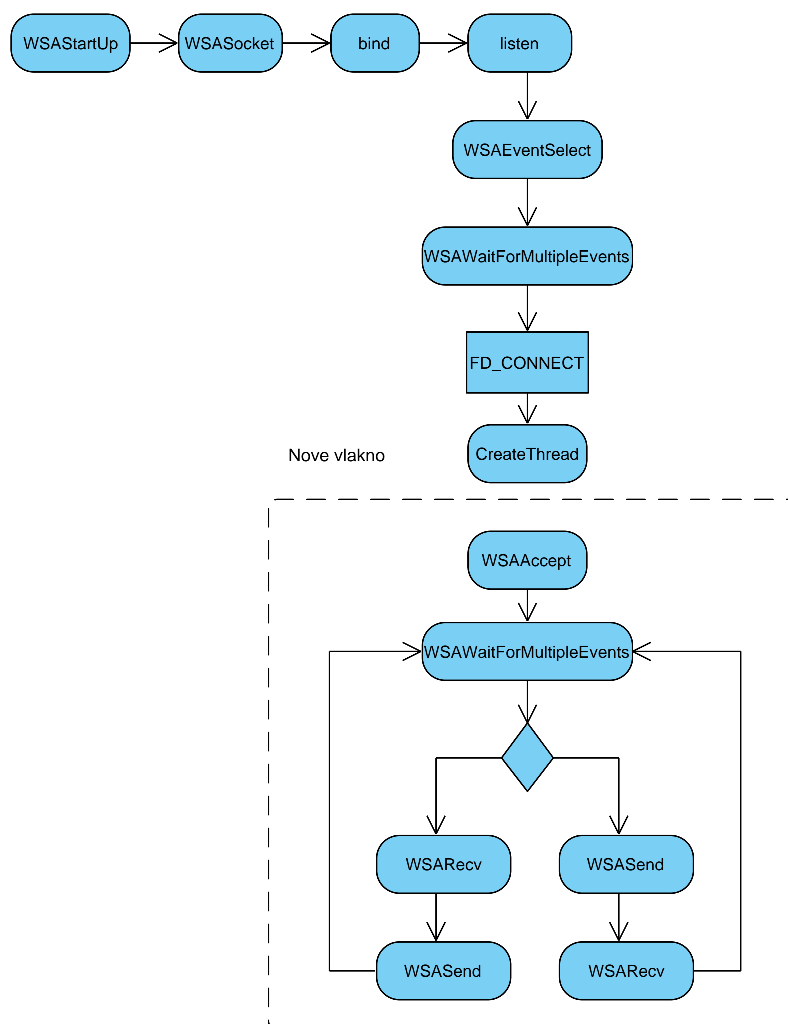
3.3.1 Návrh řešení

Server, stejně jako u předchozího řešení, inicializuje knihovnu WinSock, naslouchá na přijímacím socketu a čeká na výskyt síťové události oznamující žádost o spojení `FD_ACCEPT`. Po jejím výskytu je vytvořeno pracovní vlákno serveru funkcí `CreateThread()`. Toto vlákno pak pracuje v podstatě nezávisle na hlavní aplikaci.

V pracovním vlákně je důležité ošetřit ukončení jeho činnosti. Ideální je kontrola výskytu události, kterou hlavní aplikace signalizuje konec činnosti. K tomuto slouží funkce `WaitForSingleObject()` pro čekání na výskyt pouze jedné události, případně `WaitForMultipleObjects()` pro kontrolu výskytu více či jedné z událostí. Je možné použít i `WSAWaitForMultipleEvents()`, pokud je žádoucí zároveň kontrolovat síťové události.

Pracovní vlákno tedy čeká na výskyt jedné ze síťových událostí, případně na událost oznamující konec činnosti vlákna. Na síťové události reaguje příslušnou akcí, tedy čtením ze socketu, či zápisem dat do socketu. Je důležité poznamenat, že signalizovány jsou pouze události od klienta, pro kterého bylo vytvořeno vlákno, a ke kterém je připojen socket.

Na obrázku 3.3 je vidět jak server pracuje. Obdrží žádost o spojení (událost FD_ACCEPT), na základě této žádosti vytvoří pracovní vlákno, kterému se předají všechna potřebná data a vlákno pak obsluhuje žádosti klienta.



Obrázek 3.3: Diagram serveru vytvářejícího pro každého přijatého klienta nové vlákno.

3.3.2 Zhodnocení, problémy a nedostatky řešení

Koncepce jednoho vlákna pro každé spojení je výhodná pro implementaci serverové strany. Její režie je poměrně snadná. Stačí vytvořit vlákno a poté se postarat o korektní ukončení vlákna (výše zmíněnou signalizací události). Existuje sice omezení ze strany operačního systému co do počtu vláken, v případě serveru obsluhujícího maximálně několik stovek klientů však nehraje roli. Větším omezením může být velikost operační paměti, pokud s ní programátor a tedy jím vytvořený program nehospodaří úsporně, což obnáší uvolňování nepotřebné paměti (například uvolnění dat souvisejících s právě odpojeným klientem). Každé vytvořené vlákno potřebuje pro svou činnost určité množství operační paměti. Pokud je s ní potřebat šetřit, je možné použít koncept řízení několika vláken pomocí IOCP(Input/Outpu Completion Ports), který je popsán v následující kapitole.

3.4 Vícevláknová aplikace - řízení několika vláken pomocí IOCP

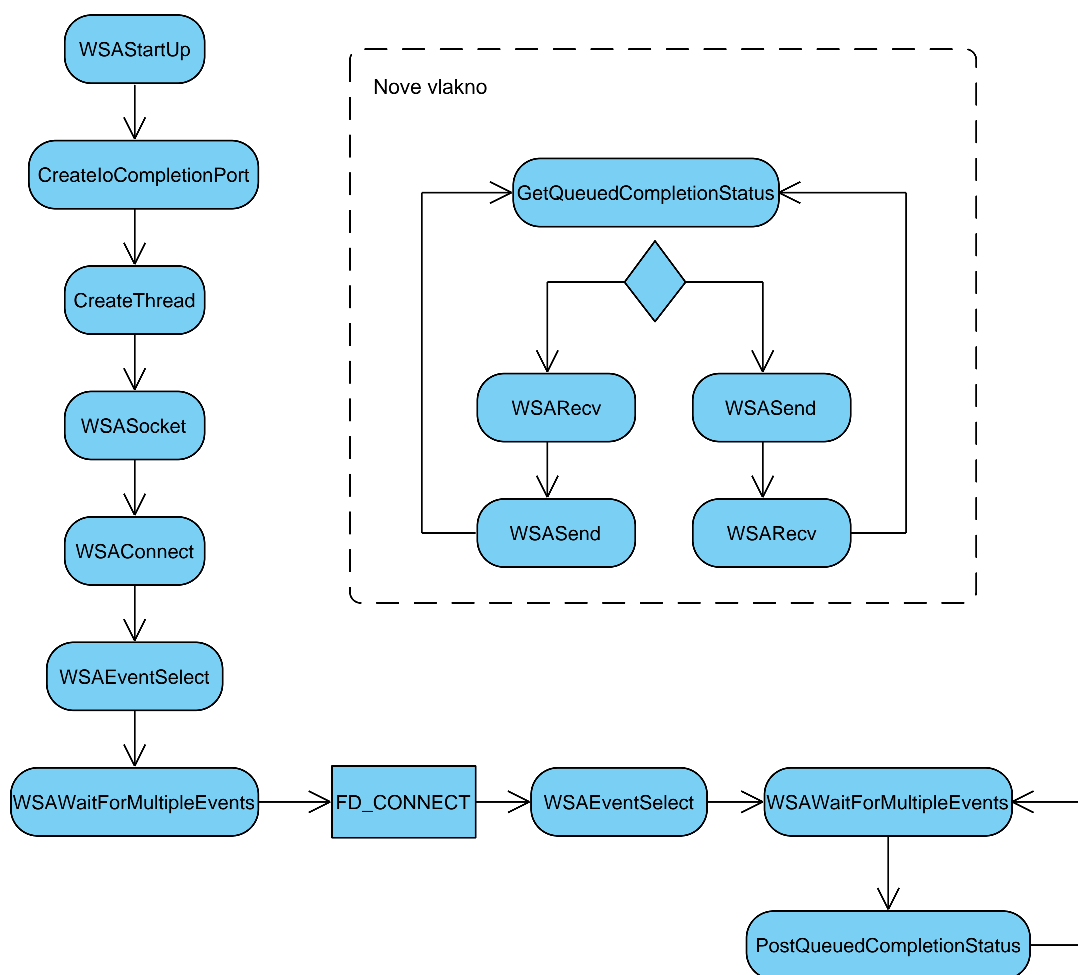
Pomocí konceptu *Input/Output Completion Ports* lze efektivně řídit běh vícevláknové aplikace. Typickým příkladem může být klientská strana komunikace, která bude komunikovat s více servery, a pro odeslání či přijem dat bude používat několik vytvořených vláken řízených pomocí IOCP.

3.4.1 Návrh řešení

Nutností je, stejně jako ve všech předchozích případech, inicializovat knihovnu WinSock. Poté je možné funkcí `CreateThread()` vytvořit tolik pracovních vláken, kolik je zapotřebí k zvládnutí požadavků aplikace. V hlavní aplikaci je však nejdříve nutné pomocí funkce `CreateIoCompletionPort()` vytvořit a inicializovat I/O Completion Port, který se předá vláknu jako parametr. Poté aplikace očekává požadavky. Takovýmto požadavkem může být například žádost k odeslání dat serveru, či výskyt síťové události, oznamující příchozí odpověď serveru. Je tedy potřeba na tyto události náležitě reagovat a do fronty IOCP požadavků přidat požadavek funkcí `PostQueuedCompletionStatus()`. Operační systém poté určí, které vytvořené vlákno si převezme požadavek a zpracuje jej.

Důležité je v kódu vlákna čekat funkcí `GetQueuedCompletionStatus()` na povel hlavní aplikace k činnosti. Poté vlákno, na základě parametru přijatého z IOCP fronty, vykoná požadovanou činnost. Buď odešle připravená data, nebo přijme data ze serveru.

Na obrázku 3.4 je znázorněno, jak takovýto klient pracuje. Jak již bylo zmíněno, je potřeba vytvořit *Completion Port* a pracovní vlákna. Dále je potřeba, pokud se jedná o spojované sockety, se připojit k serveru (či více serverům). Nakonec už nic nebrání vyhodnocování požadavků nebo síťových událostí a zařazování povelů do IOCP fronty. Jedno z vytvořených vláken pak tento požadavek převezme a provede.



Obrázek 3.4: Diagram klienta pracujícího pomocí několika vláken řízených IOCP.

3.4.2 Zhodnocení, problémy a nedostatky řešení

Jelikož koncepce Input/Output Completion Portů využívá pro svou práci pouze tolik pracovních vláken, kolik je potřeba (lépe řečeno, kolik uzná programátor za vhodné), šetří systémové prostředky, tedy operační paměť a procesorový čas. Na druhou stranu je však složitější z programátorského hlediska, protože je nutné kromě režie vláken obstarat i režii completion portu, přidávání do fronty a výběr z fronty požadavků. Poskytuje však prostředky pro vytvoření výkonné vícevláknové aplikace. Touto a předchozí etapou byl splněn třetí bod zadání.

3.5 Přenos pomocí jednotlivých protokolů

Všechny uvedené řešení pracovaly pouze s protokolem *TCP*. V následujících kapitolách budou probrány jednotlivé komunikační protokoly, způsob, jakým se komunikace pomocí

těchto protokolů implementuje, a některá specifika.

3.5.1 TCP

Protokol *TCP*, anglicky *Transmission Control Protocol* je jedním ze základních protokolů internetu. Jedná se o spojovaný protokol, což je jeho výhodou a zároveň i nevýhodou. Dva počítače mezi sebou vytvářejí spojení a protokol TCP zaručuje doručení dat ve správném pořadí, což zabraňuje použití tohoto prokolu u aplikací pracujících v reálném čase, jakou je například televize přenášená po síti. Je tedy potřeba mezi klientem a serverem vytvořit spojení a následně přenášet data. Důležité je znát adresu serveru, ke kterému se klient bude připojovat.

Na straně klienta je potřeba provést následující operace, aby přenos proběhl v pořádku:

- Funkcí `WSASocket()` vytvořit socket. Typ protokolu je potřeba nastavit na `SOCK_STREAM`.
- Naplnit strukturu `sockaddr` typem adresy (`AF_INET`), IP adresou serveru a číslem portu.
- Připojit se k serverové části pomocí funkce `WSAConnect()`, které se, mimo jiné parametry, předá vytvořený socket a struktura `sockaddr`.
- Počkat na výskyt síťové události `FD_CONNECT` oznamující přijetí spojení.
- Přijímat data funkcí `WSARecv()` nebo je odesílat pomocí `WSASend`.

Na straně serveru je potřeba provést více operací:

- Opět funkcí `WSASocket()` vytvořit socket, na kterém se bude naslouchat. Typ protokolu je potřeba nastavit stejně, tedy na `SOCK_STREAM`.
- Naplnit strukturu `sockaddr` typem adresy (`AF_INET`), IP adresou klienta, od kterého se budou přijímat data, a číslem portu. Adresu klienta lze nastavit na hodnotu `INADDR_ANY` pro příjem jakékoliv žádosti.
- Pomocí funkce `bind()` přiřadit vytvořenou strukturu naslouchacímu socketu.
- Nastavit naslouchání socketu funkcí `listen()`. Je možné omezit maximální počet čekajících žádostí o spojení, případně použít konstantu `SOMAXCONN`, pro nastavení maximálního možného počtu.
- Čekat na výskyt síťové události `FD_ACCEPT`, poté přijmout žádost o spojení pomocí funkce `WSAAccept`. Po akceptování spojení se na straně serveru vyskytne událost `FD_READ`, s čímž je nutné počítat.
- Vyhodnocovat výskyt síťových událostí `FD_READ` a `FD_WRITE` a na základě nich přijímat data funkcí `WSARecv()` nebo je odesílat pomocí `WSASend()`.

3.5.2 UDP

Protokol *UDP*, anglicky *User Datagram Protocol*, narozdíl od protokolu *TCP* není spojovaný, tudíž není potřeba se připojit od klienta k serveru. Data se jednoduše na určenou adresu odešlou. Nevýhodou je riziko, že data nedojdou v pořadí, v jakém byla odeslána. Pokud je potřeba odeslat data ve více zprávách a záleží na jejich pořadí, je potřeba programově zajistit očíslování a po přijetí následné složení zpráv do jedné.

Kromě toho je na straně klienta potřeba:

- Funkcí `WSASocket()` vytvořit socket. Typ protokolu je potřeba nastavit na `SOCK_DGRAM`.
- Naplnit strukturu `sockaddr` typem adresy (`AF_INET`), IP adresou serveru a číslem portu.
- Přijímat data funkcí `WSARecvFrom()` nebo je odesílat pomocí `WSASendTo()`. Jedním z parametrů je i vytvořená struktura.

Na straně serveru pak:

- Funkcí `WSASocket()` vytvořit socket. Typ protokolu je potřeba nastavit na `SOCK_DGRAM`.
- Naplnit strukturu `sockaddr` typem adresy (`AF_INET`), IP adresou klienta, většinou `INADDR_ANY` pro přijem dat z jakékoliv adresy a číslem portu.
- Pomocí funkce `bind()` přiřadit vytvořenou strukturu socketu.
- Vyhodnocovat výskyt síťových událostí `FD_READ` a `FD_WRITE` a na základě nich přijímat data funkcí `WSARecvFrom()` nebo je odesílat pomocí `WSASendTo()`. Jedním z parametrů je i struktura `sockaddr`. Při příjmu dat je naplněna informacemi, a je nutné ji použít pro odeslání, aby byla data správně doručena.

3.5.3 UDP broadcast

Při komunikaci typu broadcast jsou data odeslána všem počítačům v síti.

Aby klient mohl přijímat broadcastové packety, je potřeba:

- Funkcí `WSASocket()` vytvořit socket. Typ protokolu je potřeba nastavit na `SOCK_DGRAM`.
- Naplnit strukturu `sockaddr` typem adresy (`AF_INET`), IP adresou klienta, většinou `INADDR_ANY` pro přijem dat z jakékoliv adresy a číslem portu.
- Pomocí funkce `bind()` přiřadit vytvořenou strukturu socketu.
- Přijímat data funkcí `WSARecvFrom()`. Jedním z parametrů je i struktura `sockaddr`.

Server pak musí provést tyto kroky:

- Funkcí `WSASocket()` vytvořit socket. Typ protokolu je potřeba nastavit na `SOCK_DGRAM`.
- Zjistit, jestli je zapnuta podpora broadcastu funkcí `getsockopt()` s parametry `SOL_SOCKET` a `SO_BROADCAST`. Pokud je vypnuta, zapnout ji funkcí `setsockopt()` se stejnými parametry.

- Naplnit strukturu sockaddr typem adresy, typicky AF_INET, číslem portu a broadcastovou adresou. Je možné použít univerzální INADDR_BROADCAST nebo ji vypočítat způsobem uvedeným v teoretické části.
- Funkcí WSASendTo() odeslat broadcastovou zprávu. Jedním z parametrů je vytvořená struktura.

3.5.4 UDP multicast

Při komunikaci typu multicast je odeslán pouze jeden packet a to těm počítačům, které jsou přihlášeny k dané multicastové skupině. Je tedy potřeba přihlásit klienta ke skupině a odesílat data skupině počítačů.

Klient musí provést následující kroky, aby mohl přijímat data odesílaná multicastové skupině:

- Funkcí WSASocket() vytvořit socket. Typ protokolu je potřeba nastavit na SOCK_DGRAM.
- Naplnit strukturu sockaddr typem adresy (AF_INET), číslem portu a adresou INADDR_ANY, což znamená, že se budou přijímat data z jakékoliv adresy.
- Spojit vytvořenou strukturu se socketem pomocí funkce bind().
- Naplnit strukturu ip_mreq IP adresou multicastové skupiny a adresou zařízení, pomocí něhož bude probíhat komunikace s multicastovou skupinou. Základní hodnotou je INADDR_ANY.
- Funkcí setsockopt() s parametry IPPROTO_IP a IP_ADD_MEMBERSHIP a adresou vytvořené struktury přihlásit počítač k dané multicastové skupině.
- Přijímat data pomocí WSARecvFrom(), jedním z parametrů je i vytvořená struktura sockaddr.
- Po ukončení práce se odhlásit z multicastové skupiny opět funkcí setsockopt(), místo parametru IP_ADD_MEMBERSHIP je však nutné použít IP_DROP_MEMBERSHIP.

Server, který chce odesílat data multicastové skupině, musí:

- Funkcí WSASocket() vytvořit socket. Typ protokolu je potřeba nastavit na SOCK_DGRAM.
- Funkcí setsockopt() s parametry IPPROTO_IP a IP_MULTICAST_TTL a požadovanou hodnotou nastavit platnost multicastového packetu.
- Funkcí setsockopt() s parametry IPPROTO_IP a IP_MULTICAST_IF a adresou lokální stanice nastavit síťové zařízení pro odesílání packetu.
- Naplnit strukturu sockaddr typem adresy (AF_INET), číslem portu a adresou multicastové adresy.
- Odeslat data funkcí WSASendTo() multicastové skupině. Parametrem je vytvořená struktura.

3.5.5 Shrnutí

Komunikace pomocí *TCP* protokolu probíhá až po úspěšném vytvoření spojení. U komunikace založené na protokolu *UDP* se spojení nevytváří.

Při komunikaci typu *multicast* a *broadcast* se provádí spojení socketu s adresovou strukturou na straně klienta, u klasické komunikace *UDP* a *TCP* je to na straně serveru. Je to proto, že u multicastu a broadcastu jsou data očekávána na straně klienta.

Realizací podpory komunikace typu broadcast a multicast byl z jedné části splněn čtvrtý bod zadání.

3.6 Přenos dlouhých zpráv

Pokud je pro přenos dat používáno vstupně-výstupní rozhraní *overlapped*, je potřeba při přenosu dlouhých zpráv rozdělených na víc částí zajistit, aby bylo po přijetí zkontrolováno jejich pořadí. Overlapped rozhraní totiž nezajistí přenos v pořadí, v jakém byly odeslány.

Proto je do všech zpráv vložena celková délka zprávy a jednotlivé části očíslovány tak, jak jdou za sebou, a takto i odeslány. Po přijetí dat je zkontrolováno jejich pořadí a jsou správně seřazena. Díky tomu je možné přenést i soubory.

Kapitola 4

Důležité části knihovny

V této kapitole budou popsány jednotlivé třídy knihovny. Zmíněny budou i důležité funkce těchto tříd.

4.1 Třída SocketInfo

Tato třída, deklarovaná v souboru dMatrixxSocketInfo.h, představuje právě jeden socket a obsahuje všechny informace potřebné ke komunikaci pomocí tohoto socketu. Touto třídou je implementována objektová struktura správy socketů. Nejdůležitější z nich jsou:

- Socket, pomocí něhož se bude komunikovat. Před vlastní komunikací je zajištěno vytvoření tohoto socketu v dále uvedených třídách `ClientConnectionContext` nebo `ServerConnectionContext`.
- IP adresa druhé strany spojení.
- Port, na kterém probíhá komunikace.
- Typ protokolu, pomocí kterého bude probíhat komunikace. Knihovna podporuje protokoly *TCP*, *UDP* a komunikaci typu *broadcast* a *multicast* pomocí protokolu UDP.
- Strukturu `SOCKADDR_IN` pro uložení dat při komunikaci pomocí protokolu UDP.
- Události, oznamující *přijetí dat*, *odeslání dat*, *navázání spojení* a *provádění operace*.
- Mezipaměť pro uložení dat pro odeslání a přijetí.

Dále tato třída obsahuje funkce pro nastavení a získání všech těchto hodnot. Například funkce `SetBuffer()` pro nahrání dat do mezipaměti nebo `GetBuffer()` pro získání dat, typicky po skončení přijímání. Obsahuje také funkce `AddMulticastMembership()` a `DropMulticastMembership()` pro přihlášení a odhlášení stanice od multicastové skupiny.

Konstruktor a destruktor se postarají o alokaci respektive uvolnění paměti a inicializaci nebo naopak deinicializaci objektů.

Tato třída slouží k uchování proměnných potřebných pro vytvoření spojení, odeslání a přijetí dat. Neprovádí tyto operace, ale její instance je předávána funkcím, které toto provádějí.

4.2 Třída ClientConnectionContext

Třída `ClientConnectionContext`, která je deklarovaná v souboru dMatrixxSocketInfo.h, slouží k uchování kolekce socketů klientské strany tak, aby se dalo přistupovat ke každému z těchto socketů. Mimo jiné obsahuje:

- Vektor obsahující instance třídy `SocketInfo`.
- Objekt kritické sekce pro zajištění sdílení dat vektorů se sockety.
- Pole ukazatelů na události socketů. Je tak například možné čekat na dokončení odeslání dat na všechny sockety funkcí `WSAWaitForMultipleObject()`, které je jedním z parametrů předáno právě ono pole ukazatelů na jednotlivé události.
- Událost oznamující provádění operace nad všemi sockety. Tímto se liší od třídy `ServerConnectionContext`.
- Proměnnou pro uložení *HANDLE* pracovního vlákna klientské strany.

Třída také obsahuje funkce pro práci s vektorem socketů, pro přídání či odebráni socketu a smazání celého vektoru. Dále pak funkce `GetNumberOfSockets()` pro získání počtu socketů a `GetSocketInfo()` pro získání ukazatele na konrétní instanci třídy `SocketInfo`.

Konstruktor a destruktor se opět postarají o inicializaci a deinicializaci instance třídy.

Na jednom místě jsou shromážděny všechny informace o socketech aby k nim bylo možno jednoduše přistupovat.

4.3 Třída ServerConnectionContext

Třída `ServerConnectionContext`, deklarovaná v souboru dMatrixxSocketInfo.h, slouží k uchování kolekce socketů serverové strany. Jedná se v podstatě o seznam připojených počítaču. Mimo jiné obsahuje:

- Vektor obsahující instance třídy `SocketInfo`.
- Objekt kritické sekce pro zajištění sdílení dat vektorů se sockety.
- Pole ukazatelů na události socketů. Je tak například možné čekat na dokončení odeslání dat na všechny sockety funkcí `WSAWaitForMultipleObject()`, které je jedním z parametrů předáno právě ono pole ukazatelů na jednotlivé události.
- Typ protokolu, pomocí něchož se bude komunikovat. Tuto informaci je potřeba uložit při připojení klienta do nově vytvořené instance třídy `SocketInfo`.
- Socket, na kterém se bude naslouchat a přijímat spojení (při komunikaci protokolem TCP) nebo rovnou data (při komunikaci protokolem UDP).
- Port, na kterém se bude naslouchat.
- Vektor s uloženými zakázanými IP adresami.
- Pole proměnných pro uložení *HANDLE* pracovních vláken serverové strany. S tím souvisí i uložení počtu těchto vláken.

Třída obsahuje, podobně jako třída `ClientConnectionContext`, funkce pro práci s vektorem socketů, pro přídání či odebráni socketu a smazání celého vektoru. Dále pak funkce `GetNumberOfSockets()` pro získání počtu socketů a `GetSocketInfo()` pro získání ukazatele na konrétní instanci třídy `SocketInfo`.

Dále pak funkce pro prácí s vektorem zakázaných adres a zjištění, zda se požadovaná adresa nachází na černé listině.

Konstruktor a destruktor se opět postarají o inicializaci a deinicializaci instance třídy.

4.4 Pracovní vlákna a funkce pro přenos dat

V souboru dMatrixxThreadProcedures.h jsou deklarovány funkce pro přenos dat a funkce, které vykonávají vlákna. Jedná se o tyto funkce:

- Pracovní vlákno serveru `ServerWorkerThread()` starající se o chod serverové části knihovny.
- Pracovní vlákno klientské části `ClientWorkerThread()`.
- Vlákno pro vyhodnocování žádostí o spojení `AcceptThread()`.
- Vlákno `SendRecvBufferThread()` pracující pomocí konceptu IOCP a starající se o odeslání a příjem dat. Využívá níže uvedenou funkci `SendRecvBufferFunction()`.
- Funkce `AcceptConnectionFnc()`, které provede příjem žádosti o spojení.
- Funkce `SendRecvBufferFunction()` pro vykonání vlastního přenosu dat. Tato funkce na základě parametrů uložených v předané instanci třídy `SocketInfo` provede odeslání nebo přenos dat daným protokolem.
- Pro ošetření chybových stavů, především logování, slouží funkce `ErrorExit()`.
- Funkce `WriteToConsole()` se postará o logování zpráv knihovny.

4.5 Třída dMatrixxClientClass

Tato třída, deklarovaná v souboru dMatrixxClientClass.h, se stará o řízení chodu knihovny pracující jako klient. Je schopna obstarat inicializaci klienta, odeslání dat a vypnutí. Jmenovitě jsou to tyto funkce:

- O spuštění klienta se stará funkce `ClientInit()`, která inicializuje například knihovnu *WinSock* a vytvoří požadovaný počet pracovních vláken pro odeslání a příjem dat `SendRecvBufferThread()`.
- Funkce `InfoStructureInit()` sloužící k inicializaci instance třídy `ClientConnectionContext`. Tato funkce vytvoří instance třídy `SocketInfo` pro komunikaci s jednotlivými servery. Pro deinicializaci je určena funkce `InfoStructureDeinit()`.
- Funkce `CreateSocket()`, která je volána předchozí, se postará o vytvoření socketu se zadanými parametry podle požadovaného protokolu. V případě protokolu *TCP* se postará i o vytvoření spojení.

- Pro vypnutí klienta a s tím související uvolnění zabraných systémových prostředků slouží funkce `ManageShutDown()`.

4.6 Třída dMatrixxServerClass

Pro řízení chodu serverové části slouží tato knihovna. Její deklarace se nachází v souboru dMatrixServerClass.h Obdobně jako předchozí se postará o inicializaci a vypnutí, akceptování spojení a odeslání dat.

- O spuštění serveru se stará funkce `ServerInit()`, která mimo jiné inicializuje knihovnu *WinSock* a vytvoří vlákno pro akceptování dat `AcceptThread()` pro *TCP*.
- Funkce `InfoStructureInit()` sloužící k inicializaci instance třídy `ServerConnectionContext`. Do této třídy jsou přidávány jednotlivé instance třídy `SocketInfo` po akceptování spojení. Pro deinicializaci je určena funkce `InfoStructureDeinit()`.
- Funkce `BindAndListen()`, která je volána předchozí, se postará o vytvoření naslouchacího socketu se zadanými parametry podle požadovaného protokolu a o aktivování naslouchání, pokud je potřeba (při komunikaci pomocí protokolu TCP).
- Pro vypnutí serveru a s tím související uvolnění zabraných systémových prostředků slouží funkce `ManageShutDown()`.

4.7 Důležité globální proměnné a konstanty

V souboru dMatrixxSocketInfo.h jsou také definovány důležité konstanty a výčtový typ, a deklarovány globální proměnné.

4.7.1 Konstanty

Všechny konstanty jsou definovány v jednom souboru, aby byly soustředěny na jednom místě. Jmenovitě to jsou:

- `MAX_BUFFER_LEN` definující délku zprávy (mezipaměti).
- `OP_READ`, `OP_WRITE` a `OP_NOTHING` definující operace, které se budou provádět nad socketem.
- Definice konstant `WAIT_TIMEOUT_INTERVAL` definující dobu čekání serveru na spojení ze strany klienta, `SEND_TIMEOUT` dobu čekání na dokončení operace odeslání, `RECV_TIMEOUT` příjmu dat a `OPERATION_TIMEOUT` dobu čekání na dokončení ostatních operací.
- Definice konstant pro nastavení členství v multicastové skupině. Jsou to `IP_MULTICAST_IF` pro nastavení rozhraní, `IP_MULTICAST_TTL` pro omezení platnosti packetu,
  `IP_MULTICAST_LOOP` pro nastavení odeslání multicastového packetu odesílateli, `IP_ADD_MEMBERSHIP` pro přidání a `IP_DROP_MEMBERSHIP` pro odebrání členství ve skupině.

- Konstanty definující počet pracovních vláken. Jsou to CLIENT_SENDBUFFER_THREADS_PER_PROCESSOR pro nastavení počtu odesílacích a přijímacích vláken klientské části knihovny na jádro procesoru, SERVER_WORKER_THREADS_PER_PROCESSOR definující počet pracovních vláken serverové části na jádro procesoru a SERVER_ACCEPT_THREADS pro určení počtu vláken akceptujících žádosti o spojení.

4.7.2 Globální proměnné

Stejně jako konstanty, i globální proměnné jsou soustředěny na jednom místě. Jedná se o:

- Objekt kritické sekce `g_csConsole` pro sdílení přístupu ke konsoli.
- HANDLE IOCP portu k řízení vláken pro odeslání a příjem dat `g_hIOCPSendRecvBuffer` a s tím související počet konkurenčních vláken `g_iNoOfSendRecvBufferThread`.

4.7.3 Výčtový typ

V tomto souboru je také definován výčtový typ `eConnProtocol`, kterým se určuje typ protokolu, pomocí něhož knihovna komunikuje. Může nabývat těchto hodnot:

- `tcp` pro komunikaci pomocí protokolu TCP.
- `udp` pro komunikaci pomocí protokolu UDP.
- `udp_broad` pro komunikaci pomocí broadcast paketů.
- `udp_multi` pro komunikaci pomocí multicast paketů.

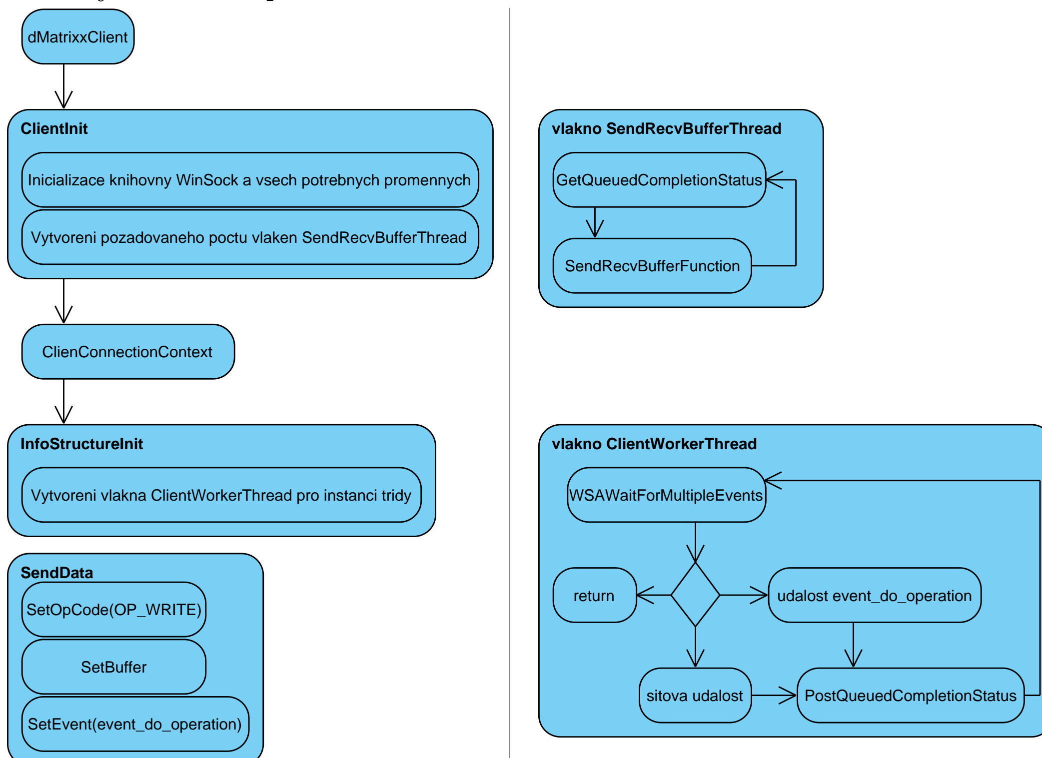
Kapitola 5

Použití knihovny

Knihovnu je možné použít jako klienta nebo server. Ve dvou následujících kapitolách bude probráno použití jak klientské, tak serverové části knihovny. Knihovna neobsahuje podporu algoritmů pro zpracování zvukových signálů v reálném čase.

5.1 Klientská část

Pro použití knihovny jako klienta je potřeba zahrnout do programu hlavičkový soubor *dMatrixxClientClass.h* a zajistit, aby se nacházel v adresáři se soubory *dMatrixxClientClass.cpp*, *dMatrixxSocketInfo.h*, *dMatrixxThreadProcedures.h* a *dMatrixxThreadProcedures.cpp*. Na obrázku 5.1 je zobrazeno použití a zároveň funkcionalita klientské části.



Obrázek 5.1: Diagram použití klienta.

Je nutné vytvořit instanci třídy dMatrixxClientClass a provést její inicializaci funkcí `ClientInit()`. Tato funkce vrací true při úspěchu, false pokud skončí chybou. V této funkci je, mimo jiné, inicializována knihovna *WinSock* a všechny potřebné proměnné. Také je vytvořen požadovaný počet vláken pro odeslání a příjem dat `SendRecvBufferThread()`, které při žádosti o přenos dat zavolá funkci `SendRecvBufferFunction()`.

Je také potřeba vytvořit instanci třídy `ClientConnectionContext` a provést její inicializaci a to funkcí `InfoStructureInit(ClientConnectionContext*, eConnProtocol, int, unsigned long, int)`, kde prvním parametrem je inicializovaná instance, druhým typ protokolu (tcp, udp, udp_broad, udp_multi), třetím port, čtvrtým pole prvků typu unsigned long s IP adresami serverů (konverzi z adresy v řetězcovém formátu lze provést pomocí funkce `GetHostIpByName()`), ke kterým se bude připojovat a posledním počet těchto prvků.

`InfoStructureInit()` je funkce třídy `dMatrixxClientClass` a kromě inicializace vyvoří i klientské pracovní vlákno `ClientWorkerThread`, které se stará o vyhodnocování síťových událostí a událostí vybízejících k provedení požadované operace.

Takto je možné inicializovat libovolné množství instancí.

Pak už jen stačí funkcí `SendData()` odeslat řetězec. Tento se postupně odešle všem adresám zadaným v inicializaci struktury. V případě úspěchu vrátí true (jinak false). Funkce uloží data do instance třídy `SocketInfo`, nastaví operaci zápisu a vyvolá událost vybízející k provedení operace.

Odeslání či přijetí dat lze provést kontrolou pole událostí `events_received_events` či `events_sent_events` nacházejících se v instanci třídy `ClientConnectionContext` pomocí funkce `WSAWaitForMultipleEvents()`.

K přijaté zprávě je možno přistupovat přes ukazatele na instanci třídy `SocketInfo`, získaného funkcí `GetSocketInfo()`, pomocí funkce `GetBuffer()` kde jediným parametrem funkce je reference na pole znaků, kam se uloží tyto data. Délku bufferu lze zjistit obdobně funkcí `GetBufferLen()`.

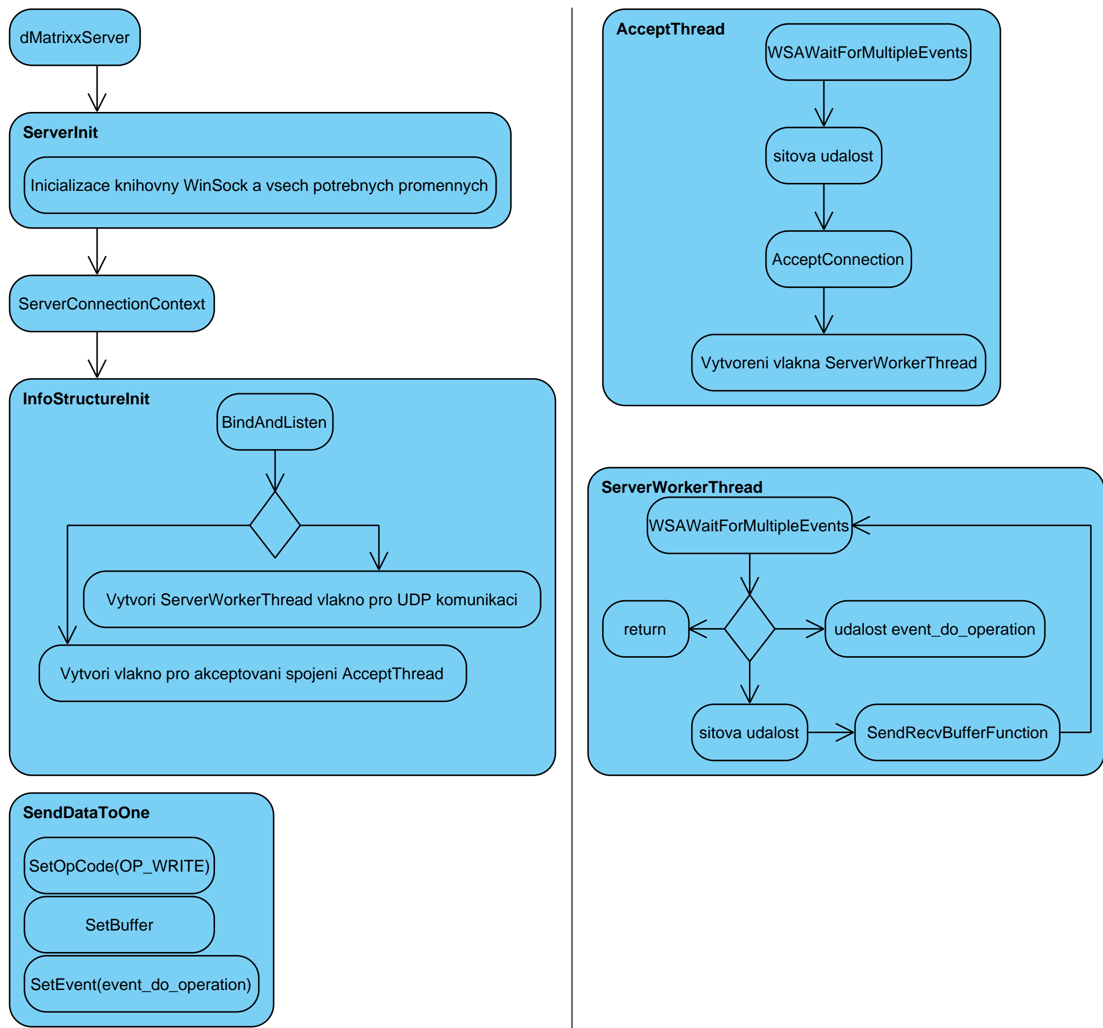
V případě komunikace pomocí UDP broadcast či UDP multicast je pouze potřeba kontrolovat výskyt událostí v poli `events_received_events`.

Klient je schopen komunikovat pomocí těchto protokolů:

- TCP - při inicializaci třídy ClientConnectionContext se nastaví výčtová hodnota `tcp`.
- UDP - výčtová hodnota `udp`.
- UDP broadcast - výčtová hodnota `udp_broad`.
- UDP multicast - výčtová hodnota `udp_multi`.

5.2 Serverová část

V případě použití knihovny jako serveru je potřeba zahrnout do programu hlavičkový soubor *dMatrixxServerClass.h* a zajistit, aby se nacházel v adresáři se soubory *dMatrixxServerClass.cpp*, *dMatrixxSocketInfo.h*, *dMatrixxThreadProcedures.h* a *dMatrixxThreadProcedures.cpp*. Na obrázku 5.2 je vidět použití a funkčnost knihovny použité jako server.



Obrázek 5.2: Diagram použití serveru.

Je potřeba vytvořit instanci třídy `dMatrixxServerClass` a provést její inicializaci funkcí `ServerInit()`. V této funkci je provedena inicializace knihovny *WinSock* a vrací true při úspěchu nebo false, pokud skončí chybou.

Dále je potřeba vytvořit instanci struktury `ServerConnectionContext` a provést její inicializaci a to pomocí funkce `InfoStructureInit(ServerConnectionContext*, eConnProtocol, int, unsigned long)` kde prvním parametrem je inicializovaná instance, druhým typ protokolu (tcp, udp, udp_broad, udp_multi), třetím port a čtvrtým hodnota adresy multicastové skupiny (pokud bude probíhat komunikace pomocí UDP multicast). `InfoStructureInit()` je funkce třídy `dMatrixxClientClass` a kromě inicializace vyvoří i vlákno pro příjem žádostí o spojení v případě komunikace pomocí protokolu *TCP*, nebo rovnou pracovní vlákno serveru `ServerWorkerThread()`. V případě komunikace pomocí protokolu *TCP* je po příjmu žádosti o spojení a ověření, že se nenachází na seznamu zakázaných adres, vytvořeno pracovní vlákno serveru `ServerWorkerThread()`, které zpracovává veškeré síťové události, které jsou indikovány na daném socketu, stejně jako událost

vybízející k provedení zadané operace.

Takto je možné inicializovat libovolné množství instancí.

Pro odeslání dat klientům je možné použít jednu z těchto funkcí:

- `SendDataToAll(const char *, ServerConnectionContext *)` - odešle zadaný řetězec všem klientům v zadané instanci třídy `ServerConnectionContext`.
- `SendDataToOne(const char *, SocketInfo *)` - odešle zadaný řetězec klientovi, na kterého ukazuje zadaný ukazatel.
- `SendDataToOne(const char *, ServerConnectionContext *, int)` - odešle zadaný řetězec klientovi ze zadané instance třídy `ServerConnectionContext`, který je v požadovaném pořadí v kolekci socketů.

Odeslání či přijetí dat lze, podobně jako u klienta, provést kontrolou pole událostí `events_received_events` či `events_sent_events` nacházejících se v instanci třídy `ServerConnectionContext` pomocí funkce `WSAWaitForMultipleEvents()`.

K přijaté zprávě je možno přistupovat přes ukazatele na instanci třídy `SocketInfo`, získaného funkcí `GetSocketInfo()`, pomocí funkce `GetBuffer()` kde jediným parametrem funkce je reference na pole znaků, kam se uloží tyto data. Délku bufferu lze zjistit obdobně funkcí `GetBufferLen()`.

Server je, podobně jako klient, schopen komunikovat pomocí těchto protokolů:

- TCP - při inicializaci třídy ClientConnectionContext se nastaví výčtová hodnota `tcp`.
- UDP - výčtová hodnota `udp`.
- UDP broadcast - výčtová hodnota `udp_broad`.
- UDP multicast - výčtová hodnota `udp_multi`.

Třída obsahuje, mimo jiné, funkci `GetIpByHostName(const char*)`, která získá z adresy serveru unsigned long hodnotu IP adresy.

Kapitola 6

Závěr

Cílem této práce bylo naprogramování knihovny pro řízení zpracování zvukových signálů v reálném čase. To obnášelo naprogramovat knihovnu, pracující na principu klient-server, která se postará o přenos potřebných dat po síti. Knihovnu, jako takovou, lze použít pro přenos jakéhokoliv typu dat. Je schopna pracovat v klientském i serverovém režimu. Komunikovat dokáže pomocí protokolů TCP a UDP a zvládá i komunikaci pomocí broadcastových a multicastových zpráv. Díky použití objektového přístupu je možné vytvořit libovolné množství kontextů, z nichž každý může komunikovat jiným protokolem a na jiném portu a odesílat tak jiná data. Knihovna tak najde využití nejen při řízení zpracování zvukových signálů. Dosud nebyla implementována podpora algoritmů pro zpracování zvukových signálů, protože ještě nebyl dokončen návrh struktury dat pro řízení zpracování.

Práce, jako taková pojednává o jednotlivých částech vytvořené knihovny, jejím použití a etapách jejího vývoje.

Za přínos této práce považuji získání spousty znalostí v oboru programování síťových aplikací a také v tvorbě vícevláknových aplikací a jejich efektivního řízení.

Uplatnění nalezne knihovna při spolupráci s firmou *Audiffex* a Martinem Slováčkem, který vypracoval bakalářskou práci na téma [12, Řízení algoritmů číslicového zpracování signálů v reálném čase pomocí HTML prohlížeče].

Možnými vylepšeními je implemetace podpory algoritmů pro zpracování v reálném čase. Jedná se o jediný bod zadání který nebyl splněn. Ve spolupráci s firmou Audiffex s.r.o. probíhá návrh struktury dat, která budou řídit vlastní zpracování.

Dalším možným vylepšením je implementace možnosti komunikace před vytvořením a ukončením spojení.

Dále pak doublebuffering, neboli dvě vyrovnávací paměti. Pokud nejsou přijatá data přečtena, je pro nově příchozí použita druhá mezipaměť a tím je zabráněno přepsání nepřevzatých dat.

Literatura

[1] 1394: *Audio and Music Data Transmission Protocol 2.1, TA Document 2001024*. 1394 Trade Association, May 24, 2002.

[2] 1394: *AV/C Digital Interface Command Set, Version 1.0*. The Multimedia Connection, September 13, 1996.

[3] cplusplus.com: C++ Library Reference. 2009.
URL http://www.cplusplus.com/reference/

[4] Fay, T.: *DirectX Audio Exposed, Interactive Audio Development*. Wordware Publishing, Inc., 2004, iSBN 1-55622-288-2.

[5] Krkavec, P.: *Využití technologie DirectInput pro řízení parametrů algoritmů číslicového zpracování signálů*. Technická zpráva, DISK Multimedia, s.r.o., 2008.

[6] MacCabe, C.: *ESBus Network Computer Management for Music-to-Picture Recording*. The 98th AES Convention, Prepring 4012 (O1).Paris, 1995.

[7] Mauro, D.: *Essential SNMP, 2nd ed*. O'Reilly Media, Inc., 2005, iSBN 0-596-00840-6.

[8] Microsoft: MSDN - Microsoft Software Development Netvork. 2009.
URL http://www.msdn.com

[9] MIDI: *The Complete MIDI 1.0 Detailed Specification, document version 96.1*. MIDI Manufacturers Association, 1997.

[10] Naugle, M.: *Illustrated TCP/IP, A Graphic Guide to the Protocol Suite*. John Wiley and Sons, Inc., 1999, iSBN 0-471-19656-8.

[11] Pirkl, J.: *Síťové programování pod Windows a programování Internetu*. Kopp nakladatelství České Budějovice CZ, 2001, iSBN 80-7232-145-5.

[12] Slováček, M.: *Řízení algoritmů číslicového zpracování signálů v reálném čase pomocí HTML prohlížeče, bakalářská práce, Brno, FEKT VUT v Brně*. 2009.

[13] Yamaha: *Yamaha mLAN Guide Book*. Yamaha Corporation, 2002.

Příloha A

Přiložené CD

CD se zdrojovými texty, programovou dokumentací a ukázkovým projektem ve vývojovém prostředí Microsoft Visual Studio 2008, demonstrujícím použití knihovny.

Struktura CD je následující

- Adresář `sources` se zdrojovými soubory knihovny.
- Adresář `text_source` obsahující zdrojový tvar písemné zprávy ve formátu
- Adresář `vsproject` s projektem vytvořeným v prostředí Microsoft Visual Studio 2008, demonstrujícím použití knihovny.
- Soubor `ReadMe.txt` s dokumentací knihovny.
- Soubor `xpilch00.pdf` obsahující písemnou zprávu.