SLEZSKÁ UNIVERZITA V OPAVĚ

Filozoficko-přírodovědecká fakulta v Opavě

# DIPLOMOVÁ PRÁCE

Opava 2016 Bc. David Havrlant

SLEZSKÁ UNIVERZITA V OPAVĚ

Filozoficko-přírodovědecká fakulta v Opavě

Bc. David Havrlant

Informatika a výpočetní technika

# Přímá komunikace v počítačové síti

# Direct Communication in a Computer Network

Diplomová práce

Opava 2016

Vedoucí diplomové práce:

RNDr. Šárka Vavrečková, Ph.D.

**Abstrakt:**

V této práci se řeší problém, který vzniká při spojování dvou počítačů (zařízení), kdy se každý z nich nachází za jiným routerem, na kterém běží NAT. Přesně taková situace nastává během spojování přes Internet.

Řešením je metoda UDP hole punching, která má dvě varianty. V textu si obě popíšeme, ale hlavně se zaměříme na náročnější z nich, u které bude sice potřeba veřejně dostupného serveru, ale pouze pro navázání spojení. Tato varianta také garantuje, že spojení bude vždy úspěšné, na rozdíl od její jednodušší varianty.

Praktickou částí je serverová aplikace, která pomáhá klientům navazovat spojení, a také klientská aplikace, která dokáže využít spojení k různým datovým přenosům jako je přenos textových zpráv, souborů, VoIP i videa.

**Klíčová slova:**

NAT, UDP hole punching, STUN server, P2P, přímá komunikace

**Abstract:**

This thesis solves the problem, which is arising during connecting two computers (devices). Both of them are hidden behind own router with NAT running on that. This situation happens during connecting through the Internet.

The UDP hole punching is method which can solve these situations. There are two variants. Thesis describes both of them, but we mainly focus on the advanced one which needs a public server but only for establishing the connection. This variant guarantees that the connection always succeeds, in contrast of its easier variant.

In the practical part there is the server application which helps clients with establish of connections. There is the client application too, which can make use of the connection to different data transfer such as text messaging, file sending, VoIP and video streaming.

**Keywords:**

NAT, UDP hole punching, STUN server, P2P, direct communication

**Prohlášení**

Prohlašuji, že jsem tuto práci vypracoval samostatně. Veškerou literaturu a další zdroje, z nichž jsem při zpracování čerpal, v práci řádně cituji a jsou uvedeny v seznamu použité literatury.

V Opavě dne …….

……..

David Havrlant

**Poděkování**

Rád bych poděkoval za odborné vedení, rady a cenné poznatky k danému tématu vedoucímu práce paní RNDr. Šárce Vavrečkové, Ph.D.

Také bych rád poděkoval mé rodině a přátelům zvláště za podporu a pomoc během mého studia.

# Obsah

# Úvod

Problémem dnešního Internetu je, že většina zařízení, která se do něj připojují, je schována za routery a nemá tedy veřejnou IP adresu, čili nejsou z Internetu adresovatelná. Hlavně je tomu z důvodu nedostatečného rozsahu IPv4. V IPv6 už je tento problém vyřešen, takže každé zařízení může mít veřejnou IP adresu, ale i přesto je možné, že uživatelé z nějakých jiných důvodů budou chtít zůstat schovaní za svými routery např. z bezpečnostních důvodů.

Pokud chce uživatel, který nemá veřejnou IP adresu, navázat spojení se serverem (počítačem), který veřejnou IP adresu má, tak nemá problém spojení navázat. To se ale nedá říct o situaci, kdy jsou dva uživatelé a ani jeden z nich nemá veřejnou IP adresu.

Možnosti jsou dvě. První variantou je nepřímá komunikace. Uživatelé mohou využít server s veřejnou IP adresou, který bude jejich komunikaci jen přeposílat, což není ideální, protože bychom si museli takový server pronajímat. Jelikož všechna data proudí přes server, tak také musí disponovat kvalitním připojením k Internetu. Také musíme počítat se zvýšením latence, což pro některé služby není vhodné. Další možností je využít veřejné servery poskytující služby jako je např. Skype. A dalším problém je bezpečnost. Poskytovatelům musíme věřit, jak s našimi daty nakládají. Skype si na serveru ukládá historii zpráv, takže důvěryhodnost zpráv je určitě na povážen ou.

Druhou variantou je přímá komunikace. V bakalářské práci jsem se možnostmi tohoto řešení zabýval. Výsledkem bylo, že takové spojení někdy navázat lze, ale nemusí k němu dojít vždy. Další informace jsou k dispozici v bakalářské práci[4]. Jedinou vždy fungující variantou, kterou se budu zabývat v této práci, je využít veřejný server, ale pouze k navázání spojení. Po něm už server není potřebný a spojení může existovat nezávisle na něm.

Součástí práce je praktická část, ve které má vzniknout aplikace pro server a aplikace pro klienty. Ta by měla využívat metody, kdy se dva nebo více uživatelů za asistence serveru vzájemně spojí a poté již mohou využívat různé služby, které aplikace bude nabízet.

Práce bude pojednávat o naprogramování obou aplikací a zaměří se i na jejich optimalizaci.

Všechny obrázky, které se v práci nacházejí, byly vytvořeny v programu Dia.

# 1 Úvodem o počítačových sítích

## 1.1 Protokoly transportní vrstvy

### 1.1.1 Protokol TCP

Protokol TCP patří do rodiny TCP/IP protokolů, konkrétně do čtvrté vrstvy, tj. transportní vrstva, referenčního modelu ISO/OSI. Jedná se o spojovaný a spolehlivý protokol. Spojovaný proto, že před samotným posílám dat se musí navázat spojení, které je možné vytvořit právě mezi dvěmi stanicemi. Po odeslání všech dat se spojení zase ukončí. A spolehlivý proto, že garantuje doručení dat ve správném pořadí. Jestliže se i přes několik znovuposlání nepodaří data doručit, dojde k ukončení spojení.

Protokolovou datovou jednotkou podle protokolu TCP je segment.[14]

### 1.1.2 Protokol UDP

Opačným protokolem k TCP je protokol UDP. Patří do stejné vrstvy jako TCP, ale nenavazuje spojení, ani není spolehlivý, což znamená, že se data odešlou, ale protokol UDP není schopen zjistit, zdali byla data doručena. Nicméně není problém řešit spolehlivost na vyšších vrstvách, pokud je potřeba, což většinou není, protože pokud chceme spolehlivě posílat data, vybereme si protokol TCP. A pokud nechceme, případné výpadky nám nevadí, protože znovuposílání ztracených dat by stejně k ničemu nebylo, neboť by se pak jednalo o stará data a ty bychom už nemohli použít.

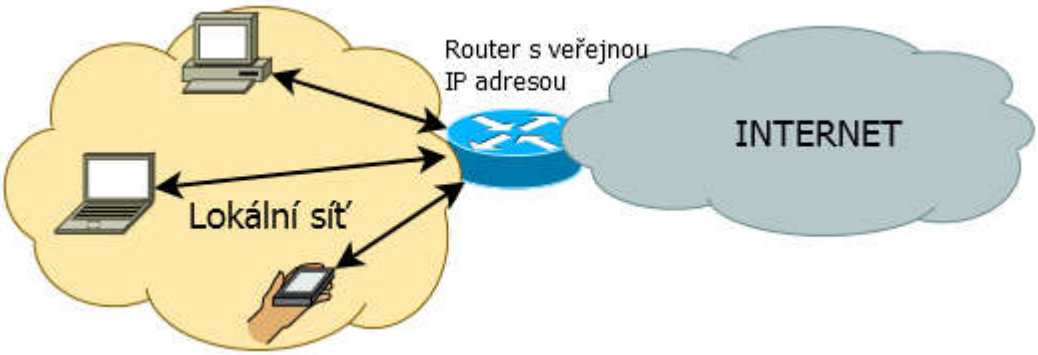
UDP má základní jednotku nazývanou datagram nebo také segment. My se však v této práci budeme držet názvu UDP datagram nebo jen datagram. UDP datagram má oproti segmentu v TCP výrazně menší hlavičku, což souvisí s jeho spolehlivostí respektive s nespolehlivostí.

Výhodou UDP proti TCP je schopnost odesílat data více stanicím pomocí multicastu nebo broadcastu. Další výhodou je větší pružnost komunikace. Během odesílání a přijímání datagramů je rychlejší jejich zpracování.[14]

## 1.2 NAT

NAT neboli překlad síťových adres vznikl jako reakce na nedostatečný počet IP adres ve verzi 4. Dalším a perspektivnějším řešením bylo IPv6, ale bylo jasné, že celý internet nepřejde na IPv6 najednou a takový proces bude trvat dlouho. Ovšem řešení muselo přijít okamžitě a tím je právě funkce NAT. Routery dnes běžně tuto funkci podporují.

Jedná se o to, že celou síť připojíme k routeru s NATem a přes něj se připojíme do Internetu. Díky tomu nepotřebujeme mít pro každý počítač veřejnou IP adresu, ale stačí nám jedna pro celou síť a tu má přiřazenou právě onen router. Obrázek 1 ukazuje, že místo tří veřejných IP adres nám bude stačit jen jedna.

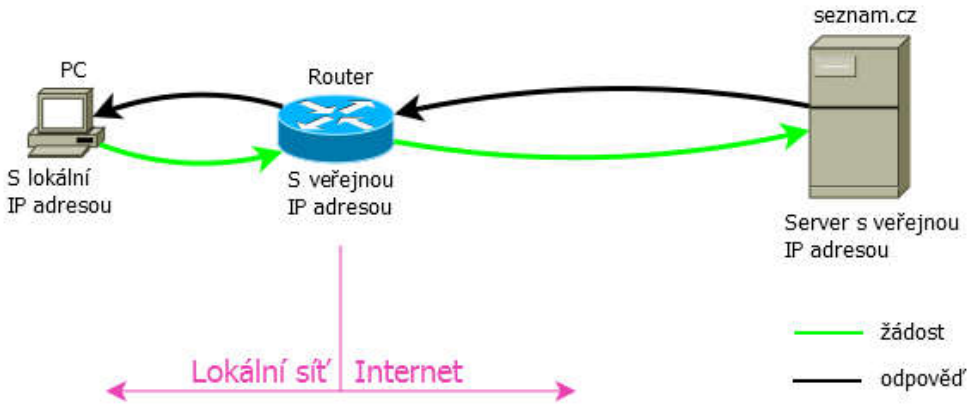



Obrázek 1 Připojení lokální sítě do Internetu

Další výhodou je, že uživatele není možné z Internetu adresovat, takže jsou chráněni. Nikdo se nemůže k jejich počítačům připojit. A pokud chtějí komunikovat, musí to být oni, kdo zinicializuje komunikaci.[19]

## 1.3 Princip fungování

Princip překladu je jednoduchý. Obrázek 2 znázorňuje běžnou situaci, kdy uživatel chce načíst webovou stránku ve svém prohlížeči.




Obrázek 2 Spojení počítače z lokální sítě se serverem s veřejnou IP adresou

Počítač posílá žádost směrem na server. Vybrané údaje z hlavičky IP paketu a k nim přidaný zdrojový a cílový port z hlavičky UDP datagramu by mohly vypadat například takto:

| Zdrojová IP adresa | Zdrojový port | Cílová IP adresa | Cílový port |
|---|---|---|---|
| 192.168.0.1 | 10001 | 77.75.76.3 | 80 |

Tabulka 1 Vybrané údaje z hlavičky paketu po odeslání z počítače

Jakmile paket dorazí do routeru a ten zjistí, že má tento paket poslat mimo vnitřní síť a tedy do Internetu, přijde ke slovu NAT. IP adresu 192.168.0.1, která je lokální, nemůže poslat do Internetu a musí ji tedy vyměnit za svou veřejnou IP adresu. Router si do své NAT tabulky připíše další záznam (ať je to zajímavější, předpokládejme, že už tam bude nějaký předchozí záznam):

| Použitý port | Zdrojová IP adresa (v LAN) | Zdrojový port (v LAN) | Cílová IP adresa | Cílový port |
|---|---|---|---|---|
| 10001 | 192.168.0.2 | 10001 | 77.75.76.3 | 80 |
| 3333 | 192.168.0.1 | 10001 | 77.75.76.3 | 80 |

Tabulka 2 NAT tabulka

Již přeložená hlavička vypadá takto:

| Zdrojová IP adresa | Zdrojový port | Cílová IP adresa | Cílový port |
|---|---|---|---|
| 90.52.3.85 | 33333 | 77.75.76.3 | 80 |

Tabulka 3 Změna hlavičky paketu

Cílová IP adresa i port zůstávají vždy stejné. Zdrojová IP adresa byla nahrazena veřejnou IP adresou daného routeru. Překvapivě se nám změnil i zdrojový port. Je to z toho důvodu, že nějaký jiný uživatel ve stejné vnitřní síti poslal žádost ze stejného zdrojového portu. Proto již tento port nemůže router využít je nucen zvolit jiný. Jakmile server zašle routeru odpověď, router právě na základě cílového portu nalezne v NAT tabulce správný záznam.[15]

# 2 UDP Hole Punching

Říkali jsme si, že počítač, který se nachází za svým routerem s NATem, není možné z Internetu adresovat. To je pravda, bez jeho spolupráce to opravdu nejde, což je správné, protože koncová zařízení to chrání před různými útoky. Co když ale máme dva uživatele, kdy každý z nich je schovaný za svým routerem, kteří se znají a chtějí mezi sebou navázat spojení s tím, že nechtějí, aby jejich komunikace procházela nějakým veřejným serverem? Právě k tomuto účelu slouží metoda UDP hole punching. Bohužel, jak již název napovídá, tuto metodu lze využívat pouze nad protokolem UDP. Sice existuje i její TCP varianta, ale ta podle mě není reálná. V zásadě jde o to, že oba uživatelé musí ve stejnou chvíli zaslat žádost o vytvoření spojení, což s latencí 50 ms není moc reálné.

Metoda UDP hole punching nám nabízí dvě možnosti. Buď ji můžeme využít úplně bez serveru, nebo jen u navázání spojení nám bude asistovat server, tzv. STUN server. Způsob řešení je ale vždy stejný. Pomocí této metody nastavíme NAT na obou routerech tak, aby komunikaci mezi dvěmi uživateli propouštěly.

## 2.1 Řešení bez využití STUN serveru

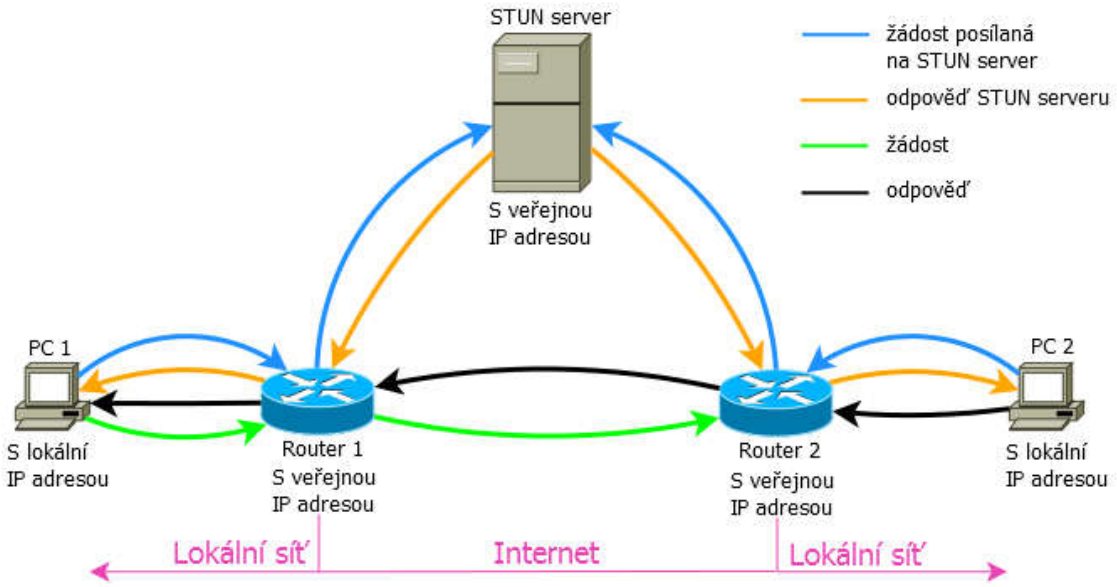
Toto řešení má jednu výhodu i jednu nevýhodu. Výhodou je, že nemusíme vlastnit nebo si pronajímat nějaký dedikovaný server. Velkou nevýhodou ale je, že úspěšnost spojení je nejistá. Jistými kroky bychom se mohli dostat na celkem vysokou pravděpodobnost úspěšnosti, ale nikdy se nebude jednat o stoprocentní metodu. Výše úspěšnosti souvisí jednak s politikou přiřazování zdrojových portů ve fázi překladu adres na routeru, ale také je ovlivněna počtem aktivním uživatelů ve vnitřní síti. V případě, že bude k Internetu připojen pouze jediný uživatel, pak router nemá důvod při překladu změnit zdrojový port. Ale v síti, kde je k Internetu připojeno mnoho uživatelů, už je šance na to, že je nějaký port zabraný jiným počítačem, mnohem vyšší.

Tímto řešením se velmi podrobně zabývám ve své bakalářské práci[4], jejíž součástí je i aplikace implementující toto řešení.

## 2.2 Řešení s využitím STUN serveru

Pokud máme dedikovaný server a na něm zprovozníme příslušnou aplikaci, pak se takový server dá nazvat STUN serverem. Ten má za úkol při spojování mezi dvěma klienty informovat každého z nich o tom druhém. Jakmile tuto činnost STUN server

vykoná, přestane být pro konkrétní spojení potřebný a toto spojení bude existovat dále nezávisle na něm. Takže bude očekávat další dva žadatele, kteří by se rádi spojili.




Obrázek 3 Navázání spojení pomocí STUN serveru

Každý klient se nejprve musí přes protokol UDP připojit ke STUN serveru, který se podívá, z jaké IP adresy a z jakého portu žádost o spojení přišla. Tyto dvě informace posílá k opačnému klientu. V této fázi se jedná klasickou komunikaci žádost-odpověď, se kterou nemá NAT žádný problém. Tabulka NAT na routeru 1 by mohla vypadat například takto:

| Použitý port | Zdrojová IP adresa (v LAN) | Zdrojový port (v LAN) | Cílová IP adresa | Cílový port |
|---|---|---|---|---|
| 3333 | 192.168.0.1 | 3333 | 90.55.8.130 | 9233 |

Tabulka 4 NAT tabulka routeru 1 při komunikaci se STUN serverem

A NAT tabulka routeru 2 takto (router 2 poslal datagram z jiného portu, než z jakého ho poslal PC 2):

| Použitý port | Zdrojová IP adresa (v LAN) | Zdrojový port (v LAN) | Cílová IP adresa | Cílový port |
|---|---|---|---|---|
| 4444 | 10.0.0.1 | 3333 | 90.55.8.130 | 9233 |

Tabulka 5 NAT tabulka routeru 2 při komunikaci se STUN serverem

V této chvíli PC 1 zná veřejnou IP adresu i port, který router přiřadil při překladu žádosti, protějšího routeru 2, a PC 2 stejně tak zná svůj protějšek. Problém je v tom, že si router v tabulce uchovává i informace o cíli. Pokud bychom poslali datagram routeru 1 na port 3333 z jiné IP adresy nebo i portu, než jakou má uvedenou v tabulce, zahodil by jej. Takže stačí přidat do tabulky routeru 1 další povolenou IP adresu a port.

Pošleme tedy z PC 1 žádost, o které víme, že ji router 2 zahodí. To nám ale nevadí, protože díky této žádosti se do NAT tabulky na routeru 1 přidá další záznam a bude vypadat následovně:

| Použitý port | Zdrojová IP adresa (v LAN) | Zdrojový port (v LAN) | Cílová IP adresa | Cílový port |
|---|---|---|---|---|
| 3333 | 192.168.0.1 | 3333 | 90.55.8.130 | 9233 |
| 3333 | 192.168.0.1 | 3333 | 97.2.67.99 | 4444 |

Tabulka 6 NAT tabulka routeru 1 po odeslání „žádosti“

Router 1 celkem logicky odeslal datagram ze stejného portu, z jakého posílal žádost STUN serveru. Nemá důvod odeslat datagram z jiného portu, když PC 1 zdrojový port také nezměnilo. V této chvíli je NAT na routeru 1 nastaven tak, že budoucí komunikaci přicházející z PC 2 propustí. Jakmile PC 2 odešle datagram směrem k routeru 1, který si bude myslet, že se jedná odpověď na jeho předešlou žádost, router 2 přidá nový záznam o další cílové adrese přiřazené ke stejnému portu do tabulky NAT. Router 2 má tedy NAT nastaven také tak, že propustí komunikaci přicházející od PC 1. Odeslaná žádost dorazí až k PC 1.

Vůbec nevadilo, že router 2 odeslal prvotní žádost STUN serveru z jiného portu než z jakého ho poslal PC 2. V řešení bez STUN serveru by tato skutečnost znamenala neúspěšné spojení.

# 3 Analýza

## 3.1 Požadavky na aplikace

Klientská aplikace, která má vzniknout v rámci plnění cílů této práce, by měla umět s pomocí STUN serveru, pro který musí existovat také nějaká aplikace, spojovat uživatele za podpory UDP komunikace a jim poté poskytovat různé služby. Nejdůležitější nabízenou službou je chat, který by měl fungovat mezi dvěma uživateli a stejně tak i mezi více uživateli. Dále by aplikace měla uživatelům umožnit mezi sebou posílat soubory, zprostředkovávat hovory, pořádat videokonference.

Důraz je kladen na to, aby spojení mezi uživateli bylo realizováno přímo, tedy nikoliv přes dedikovaný server. A to z důvodu bezpečnosti, účelem je zabránit serveru sledovat uživatele. Nicméně toto řešení s sebou přináší i nevýhody, jako problematická synchronizace historie zpráv mezi všemi zařízeními jednoho uživatele.

Tento model vyžaduje vytvoření dvou aplikací. Jedna z nich je serverová aplikace určená pro STUN server a druhá klientská aplikace. Jelikož UDP spojení nebude navazováno pouze mezi klienty, ale i mezi klienty a serverem, bude nutné zajistit spolehlivé odesílání datagramů a jejich přijímání ve správném pořadí u obou částí, tedy jak u serverové aplikace, tak i u klientské aplikace. Proto by bylo moudré vytvořit dynamicky linkovanou knihovnu, která by tuto službu poskytovala u obou dvou aplikací, abychom se vyhnuli redundantnímu kódu.

## 3.2 Existující řešení

V současné době je k dispozici mnoho aplikací, které umožňují uživatelům chatování a s ním i další doprovodné služby. Drtivá většina z nich však vždy komunikuje jen přes server. V žádném případě se nejedná o spojení klientů, které by bylo realizováno přímo.

### 3.2.1 Skype

Asi největší podobnost jsem nacházel v oblíbené aplikaci Skype. Disponuje téměř všemi mnou zmíněnými službami, kromě vzdáleného ovládání počítače, a není možné jakkoli interagovat mezi svými zařízeními, která jsou přihlášena pod stejným uživatelským účtem. Pro hovory či video hovory využívá UDP hole punching, takže mezi uživateli je pro tento druh komunikace vytvořeno spojení neprocházející žádným

serverem. Jenže bohužel textové zprávy procházejí přes server. Navíc Skype není šířen pod open-source licencí a obsahuje reklamy.[9]

### 3.2.2 Telegram

V jistém směru je podobná i aplikace jménem Telegram, která je open-source a zaměřená na bezpečnost. Je dostupná pro všechny běžné operační systémy, včetně webového rozhraní. Ve standardním režimu podporuje přihlášení z více zařízení pod společným uživatelským účtem, umožňuje vytvářet komunikace více uživatelů a umí synchronizovat historii zpráv mezi všemi zařízeními uživatelů v rámci jedné skupiny. Samozřejmostí je posílání souborů, jejichž velikost je ale limitována, což souvisí s tím, že soubory jsou nahrávány na server. Dokonce umí posílat i zvukové nahrávky. Pokud uživatelé vyloženě neprocházejí všechny možnosti v nastavení, tak se vlastně jedná o aplikaci, kterých je spousta, a která není nijak zvlášť zabezpečena.

Ale přeci jenom umí vytvořit „tajný chat“, bohužel jen na mobilním zařízení, kde už je použito end-to-end encryption a self-destruction timer. Tento způsob komunikace je ale použit právě mezi dvěma zařízeními, čili žádné jiné uživatelovo zařízení do této komunikace nepatří. Komunikace však prochází serverem, a to je vždycky riziko, a to i přes garanci end-to-end encyption. Takže záleží, jestli tvůrcům této aplikace věříme, ale jelikož se jedná o open-source projekt, tak snad by si nedovolili podvádět uživatele. A pokud jim nevěříme, máme možnost si zdrojové kódy projít a zkompilovat na svém počítači.

Volání, video konference, sdílení obrazovek ani vzdálenou plochu nepodporuje.[13]

## 3.3 Volba protokolu

Pro přímou komunikaci uživatelů jsem zvolil protokol UDP, abych mohl využít UDP hole punchingu. Otázkou bylo spojení mezi uživatelem a serverem. Na první pohled se může zdát, že nejlepším výběrem pro spojení klienta se serverem je protokol TCP. Avšak technika UDP hole punching vyžaduje i otevření UDP komunikace mezi uživatelem a serverem, aby se jednak na routeru otevřel port a jednak aby server věděl, ze kterého portu router datagram poslal. Tuto informaci potřebuje znát druhý uživatel během procesu spojování. Z tohoto důvodu jsme nuceni TCP komunikaci zavrhnout.

Data, která se zapouzdřují do UDP datagramu, mají víceúrovňovou strukturu. U všechny datagramů jsou první dva byty určeny pro uložení kanálového čísla. Nultý

kanál je však vyhrazen pro režijní data. V jeho případě určuje třetí byte typ informace jako je žádost o spojení, odpojení apod.

U nenulových kanálů jsou obecně další byty záležitostí jednotlivých typů kanálů, ale všechny mnou vytvořené typy používají pro datagramy vycházející od odesílatele další byty na identifikaci datagramů. V opačném směru bývá struktura dat různá.

O těchto pár bytech se v dalších kapitolách budu zmiňovat jako o hlavičce datagramu, rozhodně tím není myšlena skutečná hlavička UDP datagramu.

## 3.4 Volba programovacích nástrojů

### 3.4.1 C++

Pro vývoj těchto aplikací jsem zvolil můj nejoblíbenější programovací jazyk C++, a to především kvůli jeho rychlosti. Ta je zajištěna díky tomu, že programový kód se musí zkompilovat přímo pro jazyk cílového zařízení. Tento jazyk ale předpokládá, že programátor ví, co dělá. To mu ale poskytuje naprostou svobodu a může dosáhnout jakýchkoli cílů. C++ ale nedisponuje automatickou správou paměti, jak tomu je např. u Javy nebo C#, a programátor musí uvolňovat již nepotřebnou paměť ručně. Což se může jevit jako zjednodušení a tím pádem i výhoda. C#, ačkoli se stejně jako Java dá považovat za multiplatformní jazyk, nemá potřebnou rychlost C++. Také komplikovanost v něm vytvářeného kódu není prospěšná. Důvodem je, že jazyk umožňuje přepínat mezi managed a unmanaged kódem, což je skvělá vlastnost, která dovoluje volat WinAPI nebo knihovny naprogramované v C++, ale výrazně neprospívá přehlednosti kódu. Java má oproti C# o něco jednodušší kód, ale je ještě mnohem pomalejší.

C++ je multiparadigmatický programovací jazyk. Programátor si může vybrat z několika různých přístupů – procedurálního, generického či objektově orientovaného, který jsem zvolil. Nabízí obrovský počet knihoven, které programátor může při vývoji aplikace použít.

Můžeme předpokládat, že uživatelé aplikace tohoto typu ji budou mít spuštěnou rezidentně. Z tohoto důvodu je vhodné, aby zbytečně nevytěžovala CPU. Proto nejlepší možnou volbou bude právě jazyk C++.[1][7]

### 3.4.2 Visual Studio

Jedná se o vývojové prostředí, vyvinuté společností Microsoft, nejen pro vývoj C++ aplikací. K dispozici jsou varianty jak pro Windows, tak pro Linux či iOS.

Obsahuje editor kódu včetně real-timové kontroly syntaxe, debugger a samozřejmě kompilátor. Ty nejnovější verze dokonce umožňují přímo ze zdrojového kódu vygenerovat diagram tříd. Součástí je i grafický návrhář okna budoucí aplikace a spousta dalších funkcí. Toto vývojové prostředí je rychlé. Dobře a intuitivně se s ním pracuje.[17]

### 3.4.3 SQL

SQL Structured Query Language se používá v relačních databázích pro tvorbu dotazů. Při tvorbě bylo záměrem, aby dotazy byly po syntaktické stránce podobné anglickým jednoduchým větám. Tento dotazovací jazyk je stále vyvíjen. Zatím nejnovější přijatý standard je z roku 1999.[5]

Zcela určitě bude server potřebovat ukládat informace např. o uživatelích do nějaké databáze. K tomu účelu poslouží v dnešní době nejpoužívanější dotazovací jazyk SQL, který má jednoduchou syntaxi a je využíván MS SQL serverem.

### 3.4.4 Microsoft SQL Server

Microsoft již roky vydává další a další verze svého vlastního databázového systému založeného na relačním modelu.

Každý SQL server implementuje jazyk SQL mírně odlišným způsobem. Ve většině případů však základní konstrukce jazyka fungují ve všech jeho implementacích. Rozdíly se týkají spíše pokročilých dotazů. I když je otázkou, zdali např. klíčové slovo LIMIT, běžně fungující v MySQL a naopak neimplementované v Microsoft SQL, se dá brát jako pokročilé.

MS SQL je licencován jako closed source, který je v edici Express k dispozici zdarma. Oproti tomu MySQL, které je velmi populární ve webovém segmentu, používá duální model licencování. Pokud uživatel produkuje pod licencí GNU GPL verze 2, pak je MySQL zdarma, ale jestliže ne, musí využít komerční licencování. A např. PostgreSQL je šířen pod open-source licencí BSD.[2]

Tento SQL server je velmi dobře provázaný s Visual Studiem, takže výběr byl jasný.

### 3.4.5 Wireshark

Při komunikaci mezi klienty navzájem a klienty a serverem bude potřeba pro účely ladění vybrat packet sniffer.

Celosvětově velmi oblíbený nástroj, který je šířen jako open-source a do jehož vývoje se zapojil velký počet lidí, je Wireshark. Ten dokáže skenovat provoz v síti a zobrazovat jej v grafickém rozhraní (případně i v terminálu). Wireshark zná obrovské množství protokolů. Výsledná data je možné filtrovat podle různých kritérií, jako je IP adresa, port, protokol apod. Pracuje na všech úrovních síťového modelu TCP/IP a data může zachycovat z různých typů sítí jako je wifi, ethernet atd. Je vyvíjen už od roku 1998 jako multiplatformní software.[20]

Existuje i aplikace tcpdump, která ale nemá GUI a její možnosti filtrování nejsou tak dobré jako u Wiresharku. Je šířen pod BSD licencí.

Wireshark má velmi dobré využití při ladění aplikací, které jsou praktickou částí této práce, neboť je možné sledovat, zdali aplikace posílají správná data, a případně detekovat chyby aplikace.

## 3.5 Aplikace

Z analýzy vyplynulo, že v praktické části této práce bude nutné vytvořit dvě aplikace a jednu knihovnu, která by jim poskytovala síťovou podporu. Jedná se o knihovnu částečně vycházející ze síťového balíku, který je součástí mé bakalářské práce. Ten byl jednak přeprogramován z Javy do C++, ale i výrazným způsobem upraven. Cíl je podobný, musíme využít protokol UDP a na něm pak postavit spojované a v některých případech i spolehlivé spojení.

Provedení je však vedeno odlišným směrem. V kombinaci s C++ a různými optimalizacemi, kterými se v následujícím textu budu zabývat, knihovna doznala výrazného zvýšení výkonu a aplikacím poskytuje jednoduché a přitom dostačující rozhraní. Novinkou jsou různé typy komunikačních kanálů, kde je přenos dat zajištěn přesně podle typu kanálu. Spolehlivý přenos dat je vhodný ve fázi autorizace nebo při zasílání textových zpráv. Při posílání souborů je sice důležitá spolehlivost, ale i vysoká přenosová rychlost, což znamená snížení režijních dat na minimum, a to tak, že se nepotvrzuje každý přijatý datagram, nýbrž bloky dat. Naopak při streamování zvuku či videa zvolíme nespolehlivý přenos. To se velmi liší od původního řešení, kde si uživatel musel u každého datagramu zvolit, jestli ho chce přenést spolehlivě nebo nespolehlivě, ale pro účely bakalářské práce to bylo dostačující. Mechanismus kanálů s sebou přináší několik výhod, odpadá rozřazování datagramů pro různé služby, přináší prioritizační skupiny, dokáže řídit rychlost odesílání dat tak, aby se nezahltila síť

a v neposlední řadě podporuje fragmentaci dat v případě, že data by se do jednoho datagramu nevešla.

Serverová aplikace je démon, jehož úkolem je autorizovat uživatele, informovat jej o stavu ostatních uživatelů a také ostatní uživatele o nově přihlášeném.

Klientská aplikace se ovládá pomocí grafického rozhraní. To každému uživateli zobrazuje jeho seznam kontaktů, okno pro zasílání a příjímání textových zpráv s tlačítky pro hovor, odeslání souboru atd.

# 4 Knihovna pro podporu sítě

Ve výsledku se jedná o dll knihovnu, kterou si obě aplikace dynamicky přilinkují. Knihovna je naprogramována co nejobecněji to šlo, aby ji bylo možné využít i mimo tyto aplikace.

Před námi byl následující problém. Pro chatování je nejvhodnější volbou protokol TCP, který nám zajistí vše potřebné. Vytvoří spojení, udržuje ho, dokáže zajistit, že data, která pošleme, budou spolehlivě doručena, a na závěr spojení zase ukončí. TCP ale nemůžeme použít pro přímou komunikaci mezi uživateli, kteří se oba nacházejí za NAT svých routerů. Jsme tedy nuceni využít protokol UDP. Myšlenka je vytvořit simulaci TCP, postavenou na UDP, až na aplikační úrovni. Ta nám zajistí to, co by jinak zajišťoval protokol TCP sám o sobě.

## 4.1 Spojení

Oproti TCP je UDP nejelementárnější způsob, jak v sítích komunikovat, proto jej můžeme v různých situacích ohýbat podle potřeby. Například můžeme v rámci jednoho spojení komunikovat spolehlivě pří zaslání textové zprávy a zároveň nespolehlivě při uskutečňování hovoru. Kdybychom pro zaslání textové zprávy využili protokol TCP, tak bychom pro uskutečnění hovoru museli vytvářet další spojení pomocí UDP.

Spojení je vždy vytvořeno právě mezi dvěma uzly. Vzniká žádostí jednoho ze dvou zúčastněných uzlů. Po otevření spojení ještě není možné komunikovat. K tomu musí být vytvořeny komunikační kanály a přes ně pak mohou obě strany komunikovat.

V rámci spojení knihovna sleduje, zdali je protistrana stále aktivní. Pokud se jeden uzel přestane ozývat, spojení je ukončeno. Dále se měří rychlost odesílání a odezva pro každé spojení přes všechny kanály. Standardní cestou je spojení ukončeno na žádost jedné strany.

## 4.2 Komunikační kanál

Všechna spojení se skládají z kanálů, které obstarává knihovna, a které je možné vytvářet po celou dobu trvání spojení. Pokud některý uzel chce komunikovat s tím druhým, pak musí poslat žádost o vytvoření nového kanálu a s tím i informaci o typu tohoto kanálu.

Kanály umožňují pouze simplexní komunikaci. To znamená, že po vytvoření kanálu může posílat data jen ten uzel, který si kanál vytvořil. Ve skutečnosti umí kanál komunikovat ve full duplexu, ale druhý směr se využívá pouze pro řídící informace, čili data se ve druhém směru přenášet nedají.

### 4.2.1 Priority kanálů

Představme si situaci, kdy máme vytvořen jeden kanál pro přenos obrazu z kamery, další kanál pro přenos zvuku a třetí kanál pro přenos souboru. Určitě bychom nechtěli, aby nám přenos souboru zasekával přenos zvuku a obrazu. Spíše je žádoucí, aby přenos souboru využil zbývající kapacitu linky. Kdybychom si seřadili tyto služby podle důležitosti na kvalitu přenosu, což znamená kontinuální a stabilní přenos, tak bychom jako nejdůležitější pravděpodobně zvolili přenos zvuku. Dalším v pořadí by byl určitě přenos obrazu a jako poslední přenos souboru.

Priorita kanálů je realizována pomocí několika front. Tento počet se dá pomocí konstanty ve třídě *Communication* změnit. Během vytváření kanálu si uživatel může vybrat, do které fronty knihovna kanál zařadí.

Pokud by v jedné prioritní skupině byl takový kanál, který by neustále mohl odesílat data, jako je například kanál pro odesílání souborů (MassChannel), tak by se na další kanály v nižších prioritních frontách nikdy nedostalo.

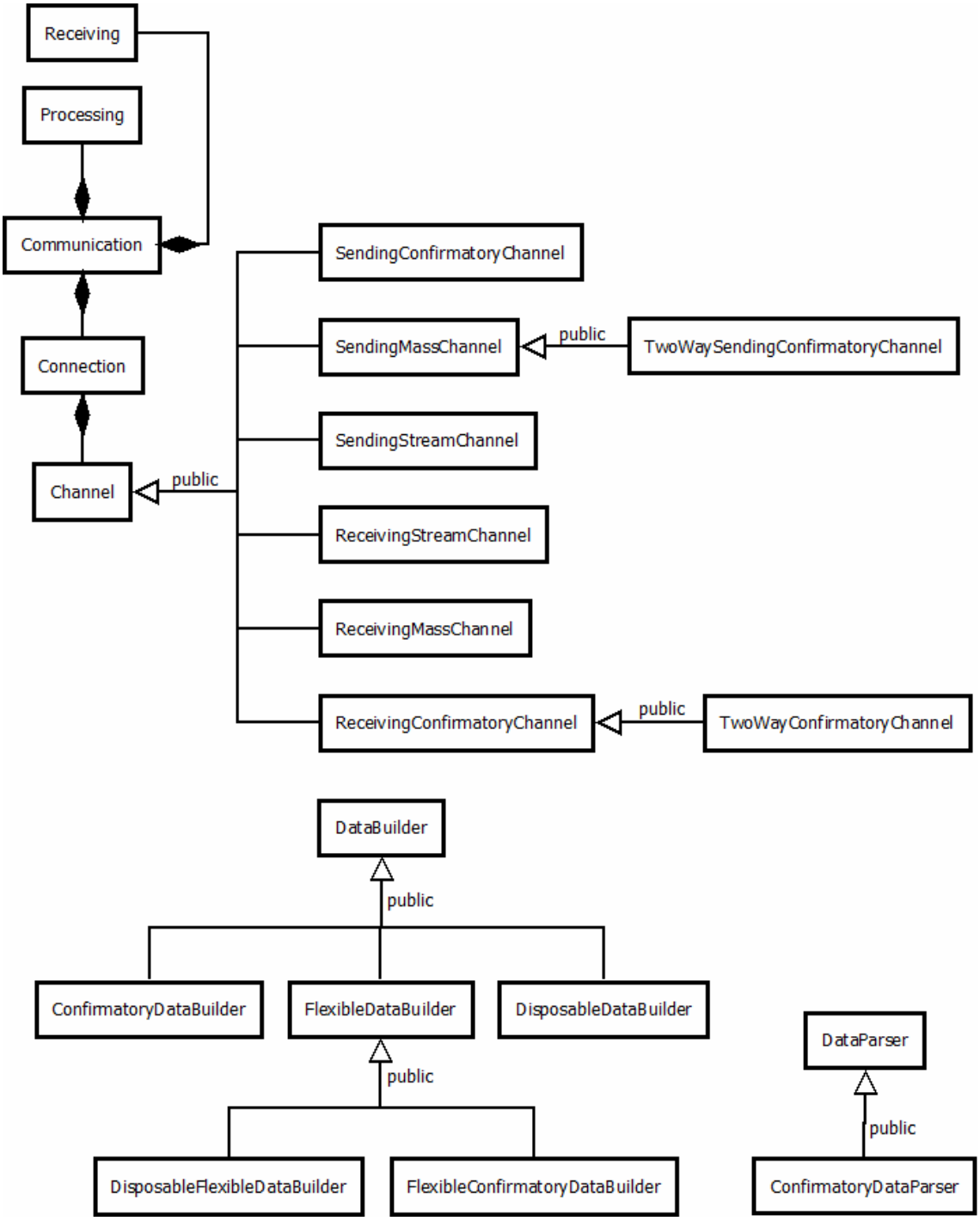
## 4.3 Základní struktura knihovny

Obrázek 4 na straně 16 znázorňuje vnitřní stavbu knihovny.

Nyní se již budeme zabývat popisem knihovny. Tím začneme v místě jednoho ze vstupů systému, tedy příjímání datagramů, na které musí systém adekvátním způsobem reagovat.

## 4.4 Příjímání datagramů

Datagramy jsou přijímány funkcí *recvfrom* nacházející se ve *WinAPI*. Ta se musí cyklicky volat, abychom mohli přijímat další a další datagramy. Po jejím zavolání se vlákno zastaví a čeká, dokud nepřijde datagram. Existuje i nastavení, ve kterém tato funkce nezastaví vlákno a ta pak vrátí buď datagram nebo nic. Tuto možnost ale my používat nebudeme.




Obrázek 4 Diagram tříd od knihovny pro podporu sítě

Důležité je, aby po vyzvednutí datagramu byla funkce *recvfrom* co nejrychleji znovu zavolána, aby tak nedošlo k přeplnění bufferu a následnému zahazování dalších datagramů. To nás přivádí k tomu, že datagramy nemůže stejné vlákno přijímat a zároveň dále zpracovávat. Jedinou možností tedy je, datagramy ukládat. O jejich zpracování se musí postarat jiné vlákno.

## 4.5 Ukládání datagramů

Nyní si musíme vysvětlit, jak datagramy co nejrychleji ukládat. Jelikož se jedná o problém producent-konzument, určitě tedy musíme použít frontu jako strukturu pro ukládání datagramů, do které přidáváme prvky na jeden konec a z druhého konce je zase odebíráme. Vhodnější bude použít dynamickou frontu, aby se nestalo, že se fronta naplní a další datagramy by byly zahazovány.

Mohli bychom využít seznam *list*, který je součástí knihovny *std*. Jedná se vlastně o dynamický seznam, jehož prvky nejsou ve fyzické paměti umístěny přímo za sebou. Proto každý jeho prvek musí obsahovat kromě dat i ukazatel na následující a předchozí prvek. Z toho vyplývá, že nemůžeme přímo přistoupit k libovolnému prvku, nýbrž musíme projít buď všechny předchozí nebo následující prvky. To nám ale nevadí, protože budeme přistupovat jen na první a na poslední prvek. Přidání i smazání se provádí v konstantním čase. Ovšem při každém vložení dalšího prvku se musí alokovat místo v paměti, a když vezmeme v úvahu frekvenci této operace, tak se nejedná o zanedbatelný čas. Vždy je výhodnější alokovat najednou více místa, než vícekrát méně místa.

Existuje výhodnější struktura *queue*, která je také součástí *std*. Ta si dokáže najedou alokovat místo pro více prvků, což už je rychlejší. Problém je však stále v tom, že nám i tato struktura provádí alokování, které je časově náročné, v kritické sekci. Což není ideální.

Proto by bylo nejlepší použít vlastní seznam, u kterého by jednotlivé prvky byly spojeny prostřednictvím ukazatelů, a to jen v jednom směru. Takže přidání dalšího prvku by bylo záležitostí přiřazení jeho adresy do ukazatele předešlého prvku s tím, že celý prvek už máme vytvořený před vstupem do kritické sekce.

Další možnou variantou by mohlo být použití struktury *queue*, ale tentokráte ne pro ukládání ukazatelů na data, nýbrž pro ukládání samotných dat. Tím bychom se vyhnuli neustálému alokování paměti pro každý přijatý datagram. Na druhou stranu by zase kód v kritických sekcích byl složitější.

### 4.5.1 Testování variant ukládání datagramů

Není jednoduché se jen tak rozhodnout pro nějakou variantu. Podívejme se tedy na výsledky testování, které nám s výběrem pomohou. V testovacím prostředí byla vytvořena třída *LinkedList*, která obsahuje dvě metody, jednu pro přidání prvku na konec fronty a druhou pro získání prvku z druhého konce. Jejich implementace se

bude v průběhu testu měnit. Dále byla vytvořena dvě vlákna. První pracuje jako producent:

```
void producent(LinkedList* linkedList)
{
	char *data;
	int j = 0;
	while (j < 1000000)
	{
		data = new char[2000];
		linkedList->add(data);
		j++;
	}
}
```

A druhé jako konzument:

```
void konzument(LinkedList* linkedList)
{
	char *data;
	int j = 0;
	while (j < 1000000)
	{
		data = linkedList->get();
		j++;
	}
}
```

Následuje popis a výsledky pro jednotlivé testované varianty ukládání datagramů.

**Fronta ukazatelů v std::queue**

Nejprve vyzkoušíme strukturu *queue* pro uchovávání ukazatelů ukazujících na přijaté datagramy.

Metoda pro přidání prvku:

```
void LinkedList::add(void* element)
{
	EnterCriticalSection(&this->cs);
	this->buffer.push(element);
	WakeConditionVariable(&this->cv);
	LeaveCriticalSection(&this->cs);
}
```

V kritické sekci do struktury *queue* s názvem *buffer* přidáme další prvek a probudíme vlákno konzumenta.

Metoda pro získání prvku:

```
void* LinkedList::get()
{
	void* data;
	EnterCriticalSection(&this->cs);
	if (this->buffer.empty())
	{
		SleepConditionVariableCS(&this->cv, &this->cs, INFINITE);
	}
	data = this->buffer.front();
	this->buffer.pop();
	LeaveCriticalSection(&this->cs);

	return data;
```

```
}
```

V kritické sekci nejdříve ověříme, jestli je fronta prázdná. V případě, že ano, vlákno se uspí a po následném probuzení už fronta prázdná nebude. Pokud ne, načteme prvek a jeho ukazatel odstraníme z fronty.

Všechny tři výsledky tohoto testu udávaly čas větší než 7 vteřin.

**Seznam Elementů**

Dále otestujeme variantu, ve které nepoužijeme žádnou strukturu z knihovny, nýbrž vlastní strukturu (v testu nazvanou *Element*), která bude disponovat ukazatelem na data a dalším ukazatelem na následující prvek. V případě, že žádný následující prvek neexistuje, bude ukazatel naplněn hodnotou NULL. Každé vlákno potřebuje svůj ukazatel (firstElement pro jedno vlákno a lastElement pro druhé vlákno) nastavit na inicializační prvek, který nebude obsahovat žádná data. Od tohoto inicializačního prvku bude jedno vlákno přidávat další prvky na jednu stranu fronty a druhé vlákno je zase bude z druhé strany odebírat.

Metoda pro přidání prvku:

```
void LinkedList::add(void* element)
{
	Element* e = new Element;
	e->data = element;
	e->next = NULL;

	EnterCriticalSection(&this->cs);
	this->lastElement->next = e;
	WakeConditionVariable(&this->cv);
	LeaveCriticalSection(&this->cs);

	this->lastElement = e;
}
```

Nejdříve vytvoříme novou instanci struktury *Element* a nasměrujeme první ukazatel struktury na data z parametru metody a druhý ukazatel ukotvíme hodnotou NULL. Poté v kritické sekci poslednímu prvku uložíme do ukazatele *next* nový následující prvek. Na konci metody ještě musíme ukazatel na poslední prvek posunout na nově přidaný prvek.

Získání prvního prvku realizuje tato metoda:

```
void* LinkedList::get()
{
	void* data;
	Element* next;

	EnterCriticalSection(&this->cs);
	if (this->firstElement->next == NULL)
	{
		SleepConditionVariableCS(&this->cv, &this->cs, INFINITE);
	}
	LeaveCriticalSection(&this->cs);
```

```
        next = this->firstElement->next;
        delete this->firstElement;
        this->firstElement = next;
        data = this->firstElement->data;

        return data;
}
```

V minulé testovací variantě jsme u metody *get* ověřovali, zdali není *buffer* prázdný. Zde ověřujeme to stejné, ale jiným způsobem. Jestliže první prvek má ukazatel *next* nastavený na NULL, znamená to, že je seznam prázdný. Pokud opustíme kritickou sekci, máme jistotu, že existuje další prvek. Dosavadní první prvek odstraníme ze seznamu, nový první prvek posuneme a načteme z něj data, která vrátíme.

Výsledky této varianty řešení byly pod 2,5 vteřinami, což je oproti předchozí variantě výrazné zrychlení.

### Fronta dat v std::queue

V poslední testované variantě vyzkoušíme strukturu *queue*, jejíž prvky budou místo předchozích ukazatelů přímo data. Data pouze obalíme do struktury *Wrapper*:

```
struct Wrapper
{
        char data[2000];
};
```

U tohoto testu je upraven kód producenta, a to tak, že data jsou alokována pouze jednou před samotným cyklem *while*.

Metoda pro přidání prvku nyní vypadá následovně:

```
void LinkedList::add(void* data)
{
        Wrapper w;
        memcpy(w.data, data, 2000);

        EnterCriticalSection(&this->cs);
        this->elements.push(w);
        WakeConditionVariable(&this->cv);
        LeaveCriticalSection(&this->cs);
}
```

Po vytvoření nové instance struktury *Wrapper* do dat nakopírujeme data předaná v argumentu metody a v kritické sekci přidáme objekt do fronty.

Metoda pro získání prvku:

```
void LinkedList::get()
{
        EnterCriticalSection(&this->cs);
        if (this->elements.empty())
        {
                SleepConditionVariableCS(&this->cv, &this->cs, INFINITE);
        }
        this->elements.pop();
        LeaveCriticalSection(&this->cs);
}
```

Po ujištění, že fronta není prázdná, jen vymažeme první prvek fronty. Data nevracíme, protože by to bylo v tomto případě komplikované a pro testovací účely je to takto dostatečné.

Výsledky tohoto testu se točí kolem hodnoty 5,8 vteřiny, což je sice lepší než první varianta, ale víc než dvojnásobně pomalejší než druhá varianta.

Výsledek testu nám dal jednoznačnou odpověď, tedy seznam Elementů je výrazně nejlepším možným řešením.

### 4.5.2 Implementace

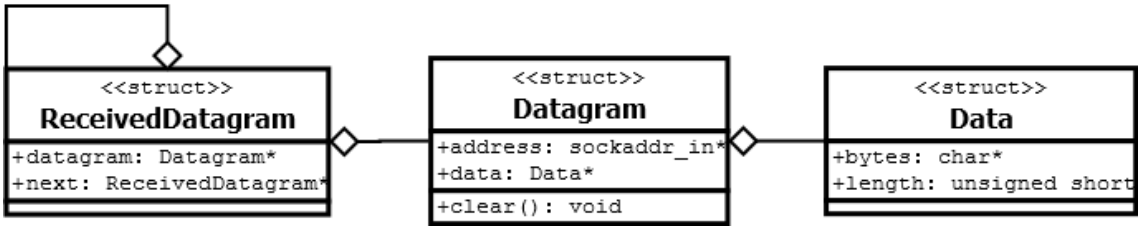
Objekt inicializovaný ze třídy *Receiving* řeší jak příjem datagramů, tak jejich ukládání. Disponuje samozřejmě vlastním vláknem. Kód vlákna vypadá následovně:

```
ReceivedDatagram* nextReceivedDatagram;
sockaddr* sourceAddress = new sockaddr;
int sourceAddressLength = sizeof(*sourceAddress);
char* buffer = new char[Communication::DATAGRAM_SIZE];
int length;
while (!this->isInterrupted)
{
        length = recvfrom(this->socket, buffer, Communication::DATAGRAM_SIZE, 0, sourceAddress,
                        &sourceAddressLength);
        if (length < 2) continue;

        nextReceivedDatagram = new ReceivedDatagram;
        nextReceivedDatagram->next = NULL;
        nextReceivedDatagram->datagram = new Datagram;
        nextReceivedDatagram->datagram->address = (sockaddr_in*) sourceAddress;
        nextReceivedDatagram->datagram->data = new Data;
        nextReceivedDatagram->datagram->data->length = length;
        sourceAddress = new sockaddr;

        if (length == Communication::DATAGRAM_SIZE)
        {
                nextReceivedDatagram->datagram->data->bytes = buffer;
                buffer = new char[Communication::DATAGRAM_SIZE];
        }
        else
        {
                nextReceivedDatagram->datagram->data->bytes = new char[length];
                memcpy(nextReceivedDatagram->datagram->data->bytes, buffer, length);
        }

        // Přidá datagram a probudí vlákno, které provede jeho zpracování
        EnterCriticalSection(&this->CS_lastReceivedDatagram); // LOCK
        this->lastReceivedDatagram->next = nextReceivedDatagram;
        WakeConditionVariable(this->CV_processing);
        LeaveCriticalSection(&this->CS_lastReceivedDatagram); // UNLOCK

        this->lastReceivedDatagram = nextReceivedDatagram;
}
```

Po počátečních inicializacích proměnných se vlákno dostane do cyklu, ve kterém se hned na začátku objevuje již zmíněná funkce *WinAPI recvfrom*. Ta je schopna vyzvednout právě jeden datagram nebo jen jeho část. V případě, že není k dispozici žádný datagram, vlákno se na tomto místě zastaví a čeká. Funkce přijímá referenci na socket, který už musí být předem vytvořen, buffer, do kterého funkce uloží přijatá

data datagramu, maximální velikost bufferu, flag, zdrojovou adresu a její velikost. Vrací počet uložených bytů do bufferu. Naplní nám buffer daty a proměnnou pro uložení zdrojové adresy adresou odesílatele. Na dalším řádku se nachází podmínka testující délku přijatých dat. Data musí být delší než jeden byte, jinak se neprovede zpracování, protože tak malý datagram není pro tento systém validní.

Poté následuje vytvoření struktur, které slouží pro uložení přijatých informací. Jejich strukturu znázorňuje následující obrázek.




Obrázek 5 Class diagram struktur pro uložení přijatého datagramu

Dále se podle velikosti přijatých dat rozhodne, jakým způsobem budou uložena. Pokud je velikost přijatých dat rovna maximální hodnotě velikosti bufferu, vytvoří se nové pole nahrazující předchozí buffer a ten je přiřazen k datagramu. Jinak se vytvoří pole, jehož velikost se rovná přijatým datům, a obsah bufferu se do něj zkopíruje.

V dalším kroku je v kritické sekci přidán na konec fronty další prvek a také je probuzeno vlákno, které seznam zpracovává. Mimo kritickou sekci se už jen posune ukazatel ukazující na poslední prvek fronty. Poté se dostáváme opět na začátek cyklu k vyčkávání na další datagram.

## 4.6 Zpracování datagramů

Nyní nám už zbývá uložené datagramy zpracovat. To se provádí ve třídě *Processing*, která také disponuje vlastním vláknem. Podívejme se tedy na zdrojový kód vlákna:

```
ReceivedDatagram* nextReceivedDatagram;

while (!this->isInterrupted)
{
        EnterCriticalSection(&this->CS_frontReceivedDatagram); // LOCK
        if (this->frontReceivedDatagram->next == NULL)
        {
                SleepConditionVariableCS(this->CV_processing, &this->CS_frontReceivedDatagram,
                        INFINITE);
        }
        LeaveCriticalSection(&this->CS_frontReceivedDatagram); // UNLOCK

        nextReceivedDatagram = this->frontReceivedDatagram->next;
        delete this->frontReceivedDatagram;
        this->frontReceivedDatagram = nextReceivedDatagram;
```

```
        this->communication->processReceivedDatagram(this->frontReceivedDatagram->datagram);
}
```

V kritické sekci zkontrolujeme, zdali je k dispozici další datagram ke zpracování. V případě, že ano, pokračujeme dál. V opačném případě se vlákno třídy *Processing* uspí, což vede k úspoře procesorového času. Jakmile vlákno pro příjem datagramů přidá do fronty přijatý datagram, ihned probudí toto vlákno. Jiný důvod probuzení vlákna není možný. Díky tomu máme jistotu, že když opouštíme kritickou sekci, máme co zpracovávat. Poté se vymaže objekt typu *ReceivedDatagram*, který slouží jako obal pro přijatý datagram, a posune se ukazatel, ukazující na začátek fronty, na následující prvek.

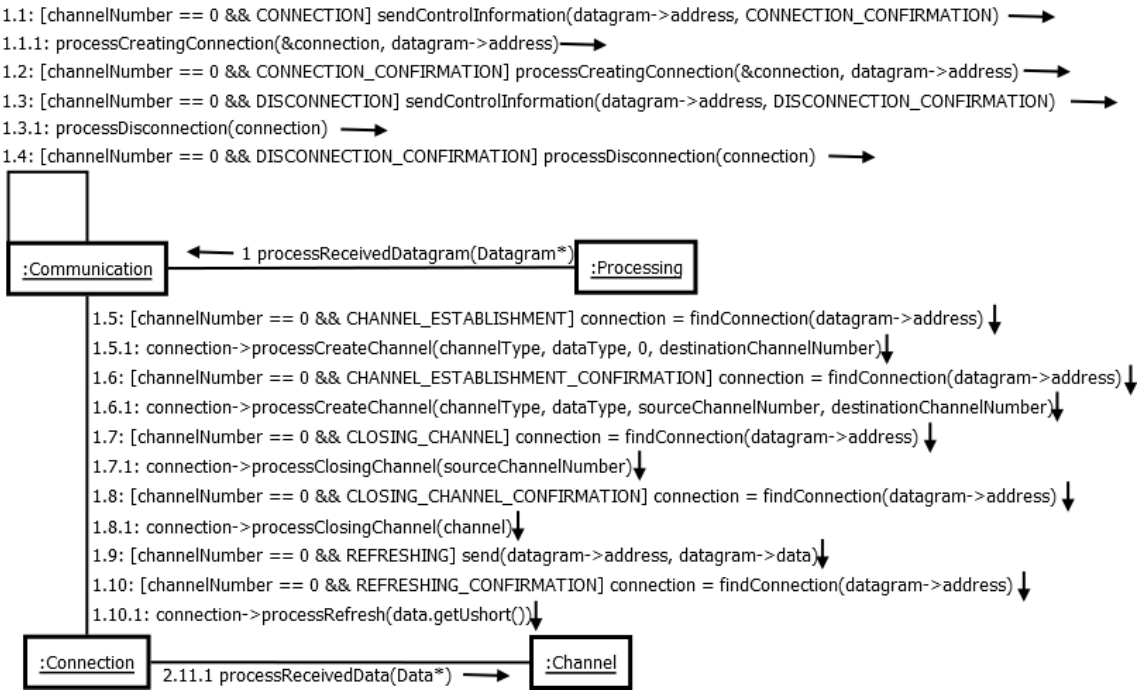
Dále je zavolána metoda třídy *Communication*, které je předán jeden datagram. Její zdrojový kód vypadá takto:

```
void Communication::processReceivedDatagram(Datagram* datagram)
{
        // Zjištění channelu
        unsigned short channelNumber = *((short*) datagram->data->bytes);

        if (channelNumber == 0) // Jedná se o řídící informaci?
        {
                if (datagram->data->length < 3) // Není datagram moc malý
                {
                        datagram->clear();
                        delete datagram;
                }

                switch (datagram->data->bytes[2]) // Typ řídící informace
                {
                        case INVITATION: break;
                        case CONNECTION: /* ... */ break;
                        case CONNECTION_CONFIRMATION: /* ... */ break;
                        case DISCONNECTION: /* ... */ break;
                        case CHANNEL_ESTABLISHMENT: /* ... */ break;
                        case CHANNEL_ESTABLISHMENT_CONFIRMATION: /* ... */ break;
                        case CLOSING_CHANNEL: /* ... */ break;
                        case CLOSING_CHANNEL_CONFIRMATION: /* ... */ break;
                        case REFRESHING: /* ... */ break;
                        case REFRESHING_CONFIRMATION: /* ... */ break;
                }
        }
        else
        {
                Connection* connection = findConnection(datagram->address);
                if (connection != NULL)
                        connection->processReceivedData(datagram->data,
                        channelNumber);
                else datagram->clear();
        }

        delete datagram;
}
```

Nejdříve se načtou první dva byty, ve kterých je uloženo kanálové číslo. Pokud se toto číslo rovná nule, jedná se o řídící informaci. V tom případě se kód rozvětví, kde se podle typu řídící informace uložené ve třetím bytu aktivuje daný případ (*case*).

V případě, že je kanálové číslo nenulové, vyhledá se spojení (*Connection*), kterému má být datagram doručen. Pokud je spojení nalezeno, předají se mu pomocí metody *processReceivedData*, která bude dále popsána, data a kanálové číslo.

Celý mechanismus znázorňuje následující obrázek.




Obrázek 6 Diagram spolupráce pro zpracování datagramů

Velmi důležitou metodou je *findConnection*, která na základě zdrojové adresy datagramu nalezne odpovídající spojení. Jakým způsobem se vyhledání provádí, bude rozebíráno v jedné z následujících kapitol.

Za zmínku stojí postupné redukování struktury začínající u *ReceivedDatagram*, která obsahuje ukazatel na strukturu Datagram. A ta obsahuje ukazatel na strukturu Data. Když zůstaneme u příkladu s předáním dat kanálu, vysvětlíme si, jak vypadá postupné odbalování těchto struktur. Před samotným zpracováním potřebujeme strukturu *ReceivedDatagram* jednak pro uložení datagramu, ale také pro uložení ukazatele na následující strukturu *ReceivedDatagram*. Jakmile tento ukazatel využijeme a uložíme si ukazatel na Datagram v ní obsažený, můžeme strukturu *ReceivedDatagram* vymazat, což se děje ještě v instanci třídy *Processing*. Ke zpracování už posíláme jen ukazatel na Datagram. Až se vyhledá náležící spojení, přestaneme potřebovat strukturu Datagram, ve které je uchována zdrojová adresa a ukazatel na strukturu Data, a můžeme ji vymazat. Až poté se kanálu předá jen struktura Data, obsahující jen bytové pole s daty, a jejich délka.

## 4.7 Operace se spojeními

Pro práci se spojeními musíme vyřešit následující operace:

1. Postupné procházení všech spojení
2. Nalezení spojení
3. Přidání spojení
4. Odstranění spojení

### 4.7.1 Analýza řešení

Vyjděme z toho, že přidávání nebo odstraňování spojení je spíše výjimečnou operací. Naproti tomu se vyhledávání spojení provádí téměř pořád. Z těchto faktů vyplývá, že vyhledávání musí být co možná nejrychlejší. To nás nutí k použití binárního vyhledávání, jenž je prováděno v logaritmickém čase. Abychom mohli používat binární vyhledávání, potřebuje mít jednotlivá spojení seřazena. Jelikož vyhledáváme podle zdrojové adresy, seřadíme tedy spojení podle IP adresy a v druhé řadě podle portu. Binární vyhledávání vyžaduje nějakou strukturu, která umožňuje přímý přístup k jakémukoli jejímu prvku. V knihovně *std* taková existuje a nazývá se *vector*.

Dalším problémem je synchronizace. Nejvhodnější bude využití SRW, kde operace 1 a 2 mohou využít sdílený mód a operace 3 a 4 exkluzivní mód.

Slim Reader Writer (SRW) Locks[10] jsou součástí Windows od verze Vista. Umožňují dva módy uzamčení, a to sdílený mód (Shared mode), ve kterém může více vláken vstoupit do kritické sekce a číst, a exkluzivní mód (Exclusive mode), ve kterém může jen jedno vlákno vstoupit do kritické sekce a modifikovat data. Takže máme dva typy funkcí:

- *AcquireSRWLockExclusive,*
- *AcquireSRWLockShared,*

a pro ukončení kritické sekce:

- *ReleaseSRWLockExclusive,*
- *ReleaseSRWLockShared.*

### 4.7.2 Realizace

**Operace 1 – postupné procházení všech spojení**

Tato operace je periodicky prováděna každých 100 ms vlastním vláknem najednou pro všechna spojení.

```
while (!this->isInterrupted)
{
	AcquireSRWLockShared(&this->LOCK_connections); // LOCK
	for (auto iterator = this->connections.begin(); iterator != this->connections.end();
		iterator++)
	{
		(*iterator)->refresh(data);
	}
	ReleaseSRWLockShared(&this->LOCK_connections); // UNLOCK
	Sleep(100);
}
```

V kritické sekci ve sdíleném módu se postupně projdou všechna spojení a zavolá se metoda *refresh*, která zajistí odeslání REFRESHING[1] datagramu a uložení aktuálního času, aby bylo možné po přijetí odpovědi vypočítat latenci.

**Operace 2 – nalezení spojení**

Každý přijatý datagram se musí přiřadit k nějakému spojení. Z toho důvodu potřebujeme metodu pro vyhledávání mezi spojeními.

Nejdříve si nadefinujeme pomocnou metodu *findConnectionIterator*, kterou budeme využívat i během dalších operací:

```
std::vector<Connection*>::iterator
	Communication::findConnectionIterator(sockaddr_in* destinationAddress)
{
	return std::lower_bound(this->connections.begin(), this->connections.end(),
		destinationAddress,
		[](const Connection* connection, sockaddr_in* address)
		{
			return (connection->destinationAddress->sin_addr.S_un.S_addr
				< address->sin_addr.S_un.S_addr
				|| (connection->destinationAddress->sin_addr.S_un.S_addr
				== address->sin_addr.S_un.S_addr
				&& connection->destinationAddress->sin_port
				< address->sin_port));
		}
	);
}
```

Ta obsahuje jen jednu funkci *lower_bound*, která se nachází v knihovně *std*, a která provádí binární vyhledávání. Té je v parametru předávána lambda funkce, ve které se objevuje poměrně komplikovaná podmínka. Ta se ale jen snaží porovnat IP adresy, a pokud by se vyskytly dvě stejné, tak se provede porovnání podle portů. Při porovnávání IP adres nejsou porovnávány postupně jednotlivé byty, nýbrž se IP adresa porovná jako 4bytové číslo.

Nyní již samotné vyhledání spojení:

```
Connection* connection = NULL;
```

[1] REFRESHING datagram v sobě nese kromě informace o svém typu také své id a používá se ke kontrole, zdali je protistrana stále na příjmu, k výpočtu latence a k prodlužování platnosti záznamu v NAT tabulce routeru v případě neaktivní komunikace.

```
AcquireSRWLockShared(&this->LOCK_connections); // LOCK
auto iterator = findConnectionIterator(destinationAddress);
if (iterator != this->connections.end()
        && compareAddresses((*iterator)->destinationAddress, destinationAddress))
        connection = *iterator;
ReleaseSRWLockShared(&this->LOCK_connections); // UNLOCK

return connection;
```

Binární vyhledávání je prováděno také v kritické sekci ve sdíleném módu. Po nalezení iterátoru se musíme ujistit, zdali se vůbec takové spojení v poli vyskytuje. Způsob tohoto ujištění souvisí s výstupy funkce *lower_bound*. Metoda pak vrátí nalezené spojení nebo NULL v případě, že žádné spojení s hledanou adresou nebylo nalezeno.

### Operace 3 – přidání spojení

Přidání nového spojení se provádí následovně:

```
AcquireSRWLockExclusive(&this->LOCK_connections); // LOCK
auto iterator = findConnectionIterator(connection->destinationAddress);

if (iterator == this->connections.end()) this->connections.push_back(connection);
else if ((*iterator) == connection)
{
        ReleaseSRWLockExclusive(&this->LOCK_connections); // UNLOCK
        return false;
}
else this->connections.insert(iterator, connection);
ReleaseSRWLockExclusive(&this->LOCK_connections); // UNLOCK

return true;
```

Abychom mohli do této struktury přidat další prvek, musíme nejdříve najít správné místo pro přidání tak, aby i poté bylo pole stále seřazeno. Celý mechanismus pracuje v kritické sekci v exkluzivním módu.

### Operace 4 – odstranění spojení

Jestliže se má spojení ukončit, musí se provést tento kód:

```
Connection* connection = NULL;

AcquireSRWLockExclusive(&this->LOCK_connections); // LOCK
auto iterator = findConnectionIterator(destinationAddress);

if (iterator != this->connections.end()
        && compareAddresses((*iterator)->destinationAddress, destinationAddress))
{
        connection = *iterator;
        this->connections.erase(iterator);
}
ReleaseSRWLockExclusive(&this->LOCK_connections); // UNLOCK

return connection;
```

Vymazání prvku musíme opět provést v kritické sekci v exkluzivním módu. Potřebujeme nalézt iterátor, který využijeme pro získání ukazatele na spojení, a také proto, abychom mohli daný prvek z pole odstranit. Ukazatel na spojení, který metodou

vracíme, posléze potřebujeme kvůli informování spojení o skutečnosti, že bude ukončeno, a dále pak k vymazání spojení samotného funkcí *delete*.

## 4.8 Operace s kanály

S kanály budeme chtít provádět tyto operace:

1. Postupné procházení všech kanálů
2. Nalezení kanálu
3. Přidání kanálu
4. Odstranění kanálu
5. Vyhledání volného kanálového čísla

### 4.8.1 Analýza řešení

**Vector**

Použitím této struktury bychom měli nejmenší možnou spotřebu místa v RAM. Pokud bychom měli prvky seřazeny podle čísla kanálu, operace 2 až 5 by byly prováděny v logaritmickém čase[1]. Jestliže by tomu tak nebylo, tak by sice přidání nového kanálu bylo prováděno v konstantním čase, ale vyhledávání až v lineárním. Když vezmeme v úvahu, že čtení a vyhledávání se v této struktuře provádí mnohonásobně vícekrát než přidávání nových, je jasnou volbou prvky seřazovat.

Zpracování bychom mohli dále urychlit lepší volbou synchronizace, a to použitím Slim Reader Writer (SRW) Locks.

Bohužel není možné získat přístup pouze pro čtení (Shared mode), vyhledat prvek, který bychom chtěli vymazat, a pak požádat o navýšení oprávnění na exclusive a prvek odstranit. Pokud bychom se to takto snažili provést, nastal by deadlock. Proto musíme jak jeho vyhledání, tak smazání provést v exklusivním módu. Problém však máme s prioritizací kanálů a s rezervováním čísla kanálu pro vytvářený kanál. Při vytváření nového kanálu potřebujeme číslo kanálu, poté ho zaslat protistraně a až po přijetí odpovědi můžeme skutečně kanál vytvořit. Jenomže během doby vyjednávání potřebujeme mít poznačeno, že kanálové číslo je již přiřazeno jistému kanálu, aby se nestalo to, že stejné číslo bude přiřazeno více kanálům.

[1] Když vezmeme v úvahu, že kanálů může být maximálně $2^{16}$, tak pro nás logaritmický čas znamená maximálně pouze 16 iterací.

**Přidání indexového pole typu vector**

Pro odstranění problému s čísly kanálů bychom mohli využit další strukturu, která by uchovávala pouze kanálová čísla. To znamená pole s prvky *unsigned short*. Tudíž velikost jednoho prvku by byla dva byty a maximální velikost celého pole by byla $2 * 2^{16}$ B = 128 KB. Což není úplně málo, ale rychlejší řešení pravděpodobně neexistuje. Nicméně existuje řešení, které by bylo mnohem přívětivější k paměti. Mohli bychom použít vector pro uchovávání čistě těch kanálových čísel, která jsou jen rezervována, ale zatím neexistuje kanál s tímto číslem. V tom však spočívá problém u algoritmu vyhledávajícího volné kanálové číslo, který pracuje v logaritmickém čase. Jedinou možností by bylo vyhledávat volné kanálové číslo v lineárním čase. Takže jak se zdá, musíme si zvolit buď rychlejší a paměťově náročnější řešení nebo naopak pomalejší, ale zase s menšími paměťovými nároky.

**Přidání více polí typu vector**

Pokud bychom chtěli přidat možnost prioritizovat kanály, nezbude nám jiná alternativa než vytvořit další vectory, jejichž počet by odpovídal počtu úrovní prioritních skupin.

**Dynamické pole místo vektoru**

S příchodem dalších struktur, kdy vectory pro prioritizaci využívá operace 1, indexové pole s kanálovými čísly využívá operace 5 a tuto strukturu využívá nyní jen operace 2, se naskytla možnost použít dynamické pole namísto původního vectoru. To by umožnilo vyhledat kanál dle jeho čísla v konstantním čase, protože index v poli by odpovídal kanálovému číslu. Ovšem v případě, kdy by byl obsazen kanál s kanálovým číslem např. 65 000 a všechny ostatní by byly volné, pak bychom museli mít v paměti alokováno 65 000 prvků pole. Takže opět řešíme problém rychlosti proti paměťovým nárokům.

### 4.8.2 Realizace

**Operace 1 – postupné procházení všech kanálů**

Potřebujeme postupně procházet všechny kanály a u každého zjistit, zdali nechce odesílat nějaká data. Což nelze realizovat jinak než lineárním vyhledáváním.

```
bool Connection::getData(Data** data)
{
        unsigned short offset, size;
        AcquireSRWLockShared(&this->LOCK_channels); // LOCK
```

```
        for (int level = 0; level < PRIORITY_LEVELS; level++)
        {
                offset = this->startInPriorityChannels[level];
                size = this->priorityChannels[level].size();
                for (auto j = 0; j < size; j++)
                {
                        if (this->priorityChannels[level][(j + offset) % size]->getData(data))
                        {
                                this->startInPriorityChannels[level] = (j + 1) % size;
                                ReleaseSRWLockShared(&this->LOCK_channels); // UNLOCK
                                return true;
                        }
                }
        }
        ReleaseSRWLockShared(&this->LOCK_channels); // UNLOCK

        return false;
}
```

Operaci 1 vykonává metoda třídy *Connection* jménem *getData*, která se snaží naplnit strukturu *Data*. Pokud se jí to podaří, vrací *true*. V případě, že žádný kanál nepotřebuje odesílat data, vrátí *false*. Algoritmus nejprve vstoupí do kritické sekce ve sdíleném módu. Poté se postupně procházejí prioritní skupiny, kdy se samozřejmě začíná od té s nejvyšší prioritou. V každé prioritní skupině se projdou všechny kanály, ale začíná se od následujícího minulého vybraného prvku, což odstraní znevýhodnění některých prvků nacházejících se na konci pole ve stejné prioritní skupině. Když některý kanál pomocí své metody *getData*(*Data*** *data*) vrátí *true* a v parametru dodá data, je procházení ukončeno, algoritmus opustí kritickou sekci a data jsou s úspěchem (*true*) předána o úroveň výš. Pokud není nalezen žádný kanál, který by chtěl odesílat data, je navrácena hodnota *false*.

### Operace 2 – nalezení kanálu

Nalezení kanálu, pro který jsou určena příchozí data, provádí následující metoda:

```
void Connection::processReceivedData(Data* data, short channelNumber)
{
        Channel* channel = NULL;

        AcquireSRWLockShared(&this->LOCK_channels); // LOCK
        channel = this->channels[channelNumber];
        if (channel != NULL) channel->processReceivedData(data);
        ReleaseSRWLockShared(&this->LOCK_channels); // UNLOCK

        if (channel == NULL) delete data;
}
```

V kritické sekci ve sdíleném módu se podle kanálového čísla proměnné *channel* přiřadí kanál z dynamického pole, jehož indexy se shodují s kanálovým číslem kanálu, který se pod ním ukrývá. Jestliže kanál s daným kanálovým číslem neexistuje, pak je proměnná *channel* naplněna hodnotou NULL. Zpracování přijatých dat je provedeno pouze v případě, že je kanál nalezen, a tedy existuje. Zpracování se také vykoná

v kritické sekci. To z toho důvodu, že by během zpracovávání mohla nastat situace, kdy by měl být stejný kanál odstraněn. Což je velmi nepravděpodobné a navíc to velmi prodlouží dobu zpracovávání v kritické sekci, ovšem je nutné podchytit všechny varianty.

**Operace 3 – přidání kanálu**

Nejdříve si nadefinujeme pomocnou metodu, která zase bude obsahovat binární vyhledávání realizované pomocí funkce *lower_bound*.

```
std::vector<Channel*>::iterator
	Connection::findPriorityChannelIterator(int priority, unsigned short channelNumber)
{
	return std::lower_bound(this->priorityChannels[priority].begin(),
		this->priorityChannels[priority].end(), channelNumber,
		[](Channel* channel, unsigned short number)
		{
			return channel->getSourceChannelNumber() < number;
		}
	);
}
```

Metoda pro přidání nového kanálu:

```
unsigned short index;

AcquireSRWLockExclusive(&this->LOCK_channels); // LOCK
this->channels.write(sourceChannelNumber, channel);

auto iterator = findPriorityChannelIterator(priority, sourceChannelNumber);
index = iterator - this->priorityChannels[priority].begin();

if (iterator == this->priorityChannels[priority].end())
	this->priorityChannels[priority].push_back(channel);
else this->priorityChannels[priority].insert(iterator, channel);

if (index <= this->startInPriorityChannels[priority]) this->startInPriorityChannels[priority]++;

ReleaseSRWLockExclusive(&this->LOCK_channels); // UNLOCK
channel->onCreate();
```

Nyní už musíme pracovat v kritické sekci v exkluzivním módu, jelikož budeme provádět modifikace. Celkem máme pro uložení kanálů tři různé struktury. Určitě musíme nový kanál přidat do jedné z prioritních skupin. Dále pak do pole jménem *channels*. Avšak ve vectoru *channelNumbers* již máme správné kanálové číslo přidané, neboť jej potřebujeme znát ještě před samotným vytvořením nového kanálu. Takže v této fázi budeme přidávat kanál jen do prvních dvou výše zmíněných struktur.

Po vstupu do kritické sekce přiřadíme do příslušného prvku pole *channels* nově vytvořený kanál. Metoda *write* zajistí případné problémy s přetečením hranice pole. Pokud by to mělo nastat, tak pole zvětší na požadovanou hodnotu. Poté se snažíme přidat ukazatel na nový kanál do jedné z prioritních skupin. To uděláme binárním

vyhledáním správného místa pro vložení tak, aby pole zůstalo seřazeno. Poté nový prvek vložíme do pole a opustíme kritickou sekci.

### Operace 4 – odstranění kanálu

Již nepotřebné kanály jsou odstraněny tímto algoritmem:

```
unsigned int index;

AcquireSRWLockExclusive(&this->LOCK_channels); // LOCK
for (int level = 0; level < PRIORITY_LEVELS; level++)
{
	auto iterator = findPriorityChannelIterator(level, channel->getSourceChannelNumber());
	index = this->priorityChannels[level].begin() - iterator;
	if (iterator != this->priorityChannels[level].end()
		&& (*iterator)->getSourceChannelNumber() == channel->getSourceChannelNumber())
	{
		this->priorityChannels[level].erase(iterator);
		if (index <= this->startInPriorityChannels[level]
			this->startInPriorityChannels[level]--;

		bool isLast = false;
		AcquireSRWLockExclusive(&this->LOCK_channelNumbers); // LOCK
		auto iterator = std::lower_bound(this->channelNumbers.begin(),
			this->channelNumbers.end(), channel->getSourceChannelNumber());
		if (iterator == --this->channelNumbers.end()) isLast = true;
		this->channelNumbers.erase(iterator);
		ReleaseSRWLockExclusive(&this->LOCK_channelNumbers); // UNLOCK

		this->channels.erase(channel->getSourceChannelNumber(), isLast);
		delete channel;

		break;
	}
}
ReleaseSRWLockExclusive(&this->LOCK_channels); // UNLOCK
```

Odstranění kanálu je velmi podobné jako jeho přidání. Opět budeme pracovat v kritické sekci v exklusivním módu. Jedinou větší změnou je to, že tentokrát už musíme pracovat i se strukturou *channelNumbers*.

Jakmile se dostaneme do kritické sekce, po jednotlivých prioritních skupinách provedeme binární vyhledávání. Pokud pozici kanálu nalezneme, odstraníme jej ze struktury *priorityChannels*. Následně budeme chtít odstranit kanálové číslo z vectoru *channelNumbers*, aby mohlo být znovu použito pro další kanály.

Důležité je, že toto odstranění provádíme v jiné kritické sekci. To proto, že tato struktura je od zbylých dvou trochu odlišná. Pracujeme s ní pouze zde a během operace 5. Navíc při zpracovávání přijatých dat (operace 2) používáme zámek *LOCK_channels* a pokud bychom chtěli vytvořit nový kanál, tak potřebujeme znát kanálové číslo. To nám řekne metoda *generateChannelNumber* (operace 5). Ale abychom se vyhnuli deadlocku, musíme pro operaci 5 zvolit jiný zámek než *LOCK_channels*.

Binární vyhledání kanálového čísla ve vectoru *channelNumbers* provedeme funkcí *lower_bound* z knihovny *std*. Před smazáním si ještě ověříme jednu podmínku. Jedná se o to, že zjistíme, jestli je odstraňovaný kanál poslední, neboli má nejvyšší obsazené kanálové číslo. To nám pomůže při smazání kanálu ze struktury *channels*. Pokud totiž byl poslední, tak se po vymazání provede test, zdali bychom nemohli pole zmenšit. V případě, že není poslední, tak se žádný test provádět nebude.

**Operace 5 – vyhledání volného kanálového čísla**

Tento mechanismus nám zajistí znovupoužitelnost kanálových čísel, než abychom pouze inkrementovali číslo. Tak bychom mohli dojít k limitu *unsigned short*, což je zhruba 65 500 hodnot, a už bychom neměli kam inkrementovat. Přitom mnoho kanálů by již bylo zcela jistě ukončeno a jejich kanálová čísla volná.

Mohli bychom použít algoritmus, u kterého by přetečení čísla nevadilo, neboť po každé inkrementaci ověří, zdali je kanálové číslo volné, a pokud není, znovu inkrementuje. Jenže ten by pracoval v lineárním čase, zatímco tento pracuje v logaritmickém.

```
unsigned short index;
unsigned short size;
unsigned short minIndex = 0;
unsigned short maxIndex;
unsigned short channelNumber;

AcquireSRWLockExclusive(&this->LOCK_channelNumbers); // LOCK
size = this->channelNumbers.size();
maxIndex = size;

if (size == 0) // pole je prázdné
{
	channelNumber = 1;
	this->channelNumbers.push_back(channelNumber);
}
else if (this->channelNumbers[size - 1] == size) // i poslední prvek není posunut
{
	channelNumber = this->channelNumbers[size - 1] + 1;
	this->channelNumbers.push_back(channelNumber);
}
else if (this->channelNumbers[0] != 1) // hned před prvním prvkem je díra
{
	channelNumber = 1;
	this->channelNumbers.insert(this->channelNumbers.begin(), channelNumber);
}
else
{
	// Binární vyhledávání
	while (true)
	{
		index = (maxIndex + minIndex) / 2;
		if (this->channelNumbers[index] == index + 1)
		{
			if (this->channelNumbers[index + 1] == index + 2) minIndex = index + 1;
			else break;
		}
		else
		{
			if (this->channelNumbers[index - 1] != index) maxIndex = index - 1;
```

```
                        else
                        {
                                index--;
                                break;
                        }
                }
        }
        channelNumber = this->channelNumbers[index] + 1;
        this->channelNumbers.insert(this->channelNumbers.begin() + index + 1, channelNumber);
}
ReleaseSRWLockExclusive(&this->LOCK_channelNumbers); // UNLOCK

return channelNumber;
```

Algoritmem se snažíme nalézt mezeru v řadě čísel (např. v řadě čísel 1, 2, 4, chybí číslo 3). Víme, že pokud bude řada čísel bez takovéto mezery, bude se např. na indexu 10 nacházet číslo 11. Ověřením této skutečnosti se pak můžeme rozhodnout, zdali řada čísel od indexu 0 do indexu 10 obsahuje nebo neobsahuje mezeru, a poté se zabývat jen levou nebo jen pravou zbývající částí řady. V počátečních podmínkách se otestují speciální případy, jako je ověření, jestli není pole prázdné. Další podmínka otestuje, zdali není nultý prvek větší než 1[1], což indikuje, že kanálové číslo 1 není obsazeno a můžeme ho ihned použít. Poslední podmínka otestuje, jestli není celá řada bez mezery, což se dá provést otestováním posledního prvku řady.

V jiném případě se provede binární vyhledávání.

## 4.9 Typy kanálů

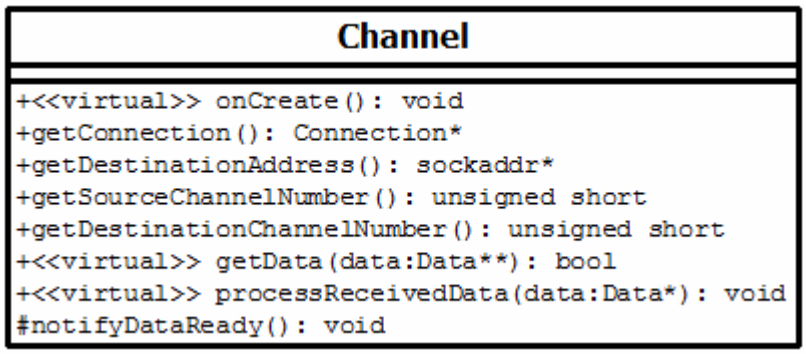
Kanál je mechanismus, který zajistí přenos dat z jedné stanice na druhou v požadované kvalitě v závislosti na jeho typu. Všechny mnou vytvořené typy kanálů mají tu společnou vlastnost, že zachovávají pořadí datagramů. Dva z nich jsou spolehlivé a jeden je nespolehlivý.

Systém je navržen tak, aby nebyl problém přidat další typ kanálu. Jednalo by se o úpravy buď přímo ve třídě *Connection* nebo vlastní třídě z ní zděděné, do které by se musely přidat dvě čistě virtuální metody (jedna pro odesílání a druhá pro příjem), a do virtuální metody *creatingChannel* přidat další větev *case*. Jelikož je metoda virtuální, tak by se mohla ve zděděné třídě vytvořit metoda pojmenovaná stejně, do ní přidat potřebné případy (*casy*), ale na konci metody nezapomenout zavolat původní metodu, aby původní funkčnost zůstala zachována.

Každý typ kanálu musí zdědit třídu *Channel* a pak na základě požadavků na něj kladených může překrýt tři virtuální metody. Virtuální metoda *onCreate* je zavolána

[1] 1 je první povolená hodnota pro kanálové číslo, 0 je vyhrazena pouze pro režijní řízení

ihned po vytvoření kanálu. Další metodou je *getData(Data** data)* vracející *bool*, která je volána v rámci procházení všech kanálů a zjišťování, zdali některý nechce odesílat data. Metoda vrací *false*, pokud nechce nic odesílat, anebo *true* v případě, že chce, a přes argument předá data, která chce poslat. Data nebudou smazána. O jejich smazání se musí postarat kanál. Poslední metodou je *processReceivedData(Data* data)*, která uživateli předává přijatá data, o jejichž smazání se musí postarat kanál. Obrázek 7 situaci zpřehledňuje.




Obrázek 7 Zjednodušený class diagram třídy Channel

Kanály mohou využívat metody nacházející se ve třídě *Channel*, jako je *notifyDataReady*, která probudí vlákno, které se ptá všech kanálů, zdali nechtějí odesílat data. Vlákno se může uspat v případě, že žádný kanál vysílat nechtěl. K dispozici jsou i další metody na zjištění adresy, zdrojového či cílového kanálového čísla nebo spojení, ve kterém se kanál nachází. V neposlední řadě se zde nachází metoda *close* pro uzavření kanálu.

### 4.9.1 ConfirmatoryChannel

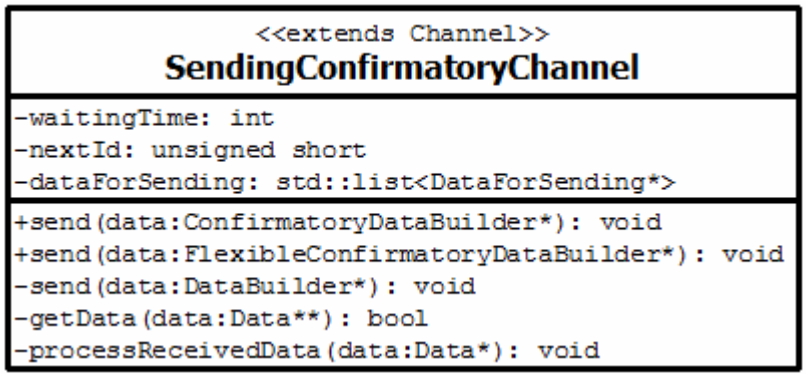
Tento typ kanálu zprostředkovávající veškerou komunikaci mezi klientem a serverem je ze všech tří typů tím nejzákladnějším. Také se pomocí něj posílají zprávy mezi uživateli. Jsou využity všude kromě přenosu souborů, videa nebo audia.

Každý datagram musí být potvrzen, jinak jej po určitém časovém úseku odesílatel pošle znovu, a to se opakuje tak dlouho, dokud jej příjemce nepotvrdí. Další datagramy se sice odesílají, ale kanálové rozhraní u příjemce datagramy ukládá do paměti, aby se zajistilo jejich správné pořadí. Až jakmile přijde chybějící datagram, spolu s jeho předáním uživateli knihovny se předají všechny ostatní datagramy, které byly uchovány v paměti.

ConfirmatoryChannel podporuje i posílání datagramů větších než povolený limit nastavený v aplikaci, čehož je dosahováno fragmentováním nadlimitních datagramů.

**SendingConfirmatoryChannel**

Odesílání datagramů probíhá tak, že uživatel volá metodu *send*, která do seznamu uloží datagram nebo více datagramů, pokud by jeden byl větší než povolená hranice a musel by se fragmentovat. Obrázek 8 znázorňuje strukturu této třídy.




Obrázek 8 Zjednodušený class diagram třídy SendingConfirmatoryChannel

Metoda *send(DataBuilder* data)* je realizována takto:

```
unsigned int remainingSize = data->getLength();
unsigned short size;
bool isThisFirstFragment = true;
bool isThisLastFragment;

while (remainingSize > 0)
{
	DataForSending* dataForSending = new DataForSending;
	if (data->getLength() > Communication::DATAGRAM_SIZE) // Pro velké datagramy
	{
		if (remainingSize > Communication::DATAGRAM_SIZE - 4)
			size = Communication::DATAGRAM_SIZE - 4;
		else
		{
			size = remainingSize;
			isThisLastFragment = true;
		}

		dataForSending->data = new Data(size + 4);
		memcpy(dataForSending->data->bytes + 4,
			data->getData()->bytes + 4 + data->getLength() - remainingSize,
			size);
		if (isThisLastFragment) delete data;
	}
	else // Pro malé datagram, aby se zbytečně nekopírovaly
	{
		dataForSending->data = data->getData();
		size = data->getLength();
		isThisLastFragment = true;
	}

	// Nastavení hlavičky datagramu
	dataForSending->data->bytes[0] = getDestinationChannelNumber();
	dataForSending->data->bytes[1] = getDestinationChannelNumber() >> 8;
	dataForSending->data->bytes[2] = this->nextId;
	dataForSending->data->bytes[3] = (this->nextId >> 8) & 0x3f; // Ořežeme první 2 bity
	if (isThisFirstFragment) dataForSending->data->bytes[3] |= 0x80;
```

```
            if (isThisLastFragment) dataForSending->data->bytes[3] |= 0x40;
            isThisFirstFragment = false;
            this->nextId += 1;

            EnterCriticalSection(&this->CS_dataForSending); // LOCK
            this->dataForSending.push_back(dataForSending);
            LeaveCriticalSection(&this->CS_dataForSending); // UNLOCK

            remainingSize -= size;
}

notifyDataReady();
```

Do proměnné *remainingSize* si uložíme celkovou délku dat. Následuje cyklus *while*, který se bude provádět, dokud bude proměnná *remainingSize* větší než nula, čili bude co odesílat. V cyklu nejdříve inicializujeme strukturu *DataForSending*, poté vyhodnotíme podmínku, kterou zjišťujeme, zdali jsou data moc velká anebo dostatečně malá na to, abychom je mohli odeslat jen v jednom datagramu. Do proměnné *size* si načteme buď hodnotu maximální velikosti datagramu nebo menší hodnotu. S nastavením menší hodnoty souvisí i to, že tento fragment bude posledním. Následně si alokujeme místo pro uložení dat a ta pak do něj zkopírujeme ze zdrojových dat z patřičné pozice, kterou v každém cyklu posouváme. Jestli se již jedná o poslední fragment, můžeme vymazat zdrojová data, ze kterých jsme kopírovali.

V případě, že se jednalo o dostatečně malá data, pouze na ně nastavíme ukazatel. U velkých dat bylo kopírování nezbytné, protože jsme potřebovali do každého dalšího datagramu přidat také hlavičku. Na rozdíl od malých dat, kde už je místo pro hlavičku rezervováno. To je také důvod toho, že kopírovat začínáme až čtvrtý byte.

Poté nastavíme hlavičku datagramu a to tak, že do prvních dvou bytů vložíme číslo cílového kanálu a do dalších dvou uložíme ořezané identifikační číslo datagramu. Ořezáním získáme dva první bity nulové. Ty použijeme k nastavení fragmentu. První bit nastavený na jedničku značí, že daný fragment je prvním v řadě. Druhý bit značí, že je posledním v řadě. Ostatní fragmentované datagramy mají nastavené oba bity na nulu. Nefragmentované datagramy mají naopak nastavené oba bity na jedničku, protože jsou de facto prvním a zároveň posledním v řadě.

Takto připravený datagram vložíme do seznamu, jehož vlastní odeslání zařídí jiné vlákno, poté ještě z proměnné *remainingSize* odečteme délku tohoto kusu dat. Nakonec metodou *notifyDataReady* informujeme vlákno odesílající datagramy, že máme data k odeslání a může si je vyzvednout.

Vyzvednutí dat a následné odeslání zajišťuje metoda *getData(Data** data)*:

```
EnterCriticalSection(&this->CS_dataForSending); // LOCK
for (DataForSending* dataForSending: this->dataForSending)
{
```

```
        if ((dataForSending->attempts == 0)
                || (currentTime - dataForSending->lastAttempt > this->waitingTime))
        {
                *data = dataForSending->data;
                dataForSending->attempts++;
                dataForSending->lastAttempt = currentTime;
                LeaveCriticalSection(&this->CS_dataForSending); // UNLOCK
                return true;
        }
}
LeaveCriticalSection(&this->CS_dataForSending); // UNLOCK

return false;
```

V kritické sekci se prochází seznam, do kterého v předchozí metodě ukládáme připravené datagramy. Seznam se prochází od počátku, ale první datagram nemusí být vybrán. Vybrán je takový, který zatím ještě nebyl odeslán nebo sice již byl odeslán, ale dávno.

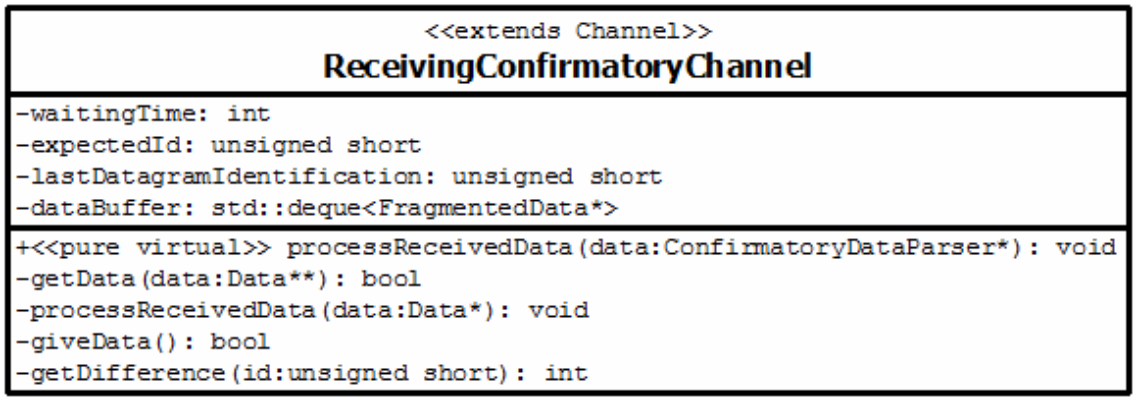
Poslední metoda *processReceivedData(Data* data)*, která ošetřuje příchod datagramů, vypadá následovně:

```
DataForSending* dataForSending;
EnterCriticalSection(&this->CS_dataForSending); // LOCK
for (auto iterator = this->dataForSending.begin(); iterator != this->dataForSending.end();
        iterator++)
{
        dataForSending = *(iterator);
        if (*((short*) (dataForSending->data->bytes + 2)) == *((short*) (data->bytes + 2)))
        {
                this->dataForSending.erase(iterator);
                delete dataForSending;
                break;
        }
}
LeaveCriticalSection(&this->CS_dataForSending); // UNLOCK

delete data;
```

Opět v kritické sekci procházíme seznam a hledáme takový datagram, jehož id se shoduje s číslem, které uloženo v příchozím potvrzovacím datagramu. Poté vymažeme nejen samotný datagram, ale také ukazatel v seznamu. Nakonec odstraníme potvrzovací datagram.

### ReceivingConfirmatoryChannel

Obrázek 9 ukazuje strukturu této třídy.




Obrázek 9 Zjednodušený class diagram třídy ReceivingConfirmatoryChannel

Příjem datagramů zajišťuje metoda *processReceivedData(Data* data)*:

```
// Pošleme potvrzení o přijetí
char* bytes = new char[4];
bytes[0] = getDestinationChannelNumber();
bytes[1] = getDestinationChannelNumber() >> 8;
bytes[2] = data->bytes[2];
bytes[3] = data->bytes[3];
Datagram confirmation(getDestinationAddress(), bytes, 4);
getConnection()->getCommunication()->send(&confirmation);
delete bytes;

// Obalíme další strukturou
FragmentedData* fragmentedData = new FragmentedData;
fragmentedData->data = data;
fragmentedData->isThisFirstFragment = data->bytes[3] >> 7;
fragmentedData->isThisLastFragment = (data->bytes[3] & 0x7f) >> 6;
data->bytes[3] = data->bytes[3] & 0x3f;

// Zpracování
unsigned short id = *((unsigned short*) (data->bytes + 2));
int difference = getDifference(id);
if (difference >= 0)
{
        auto iterator = std::lower_bound(this->dataBuffer.begin(), this->dataBuffer.end(), id,
                [](FragmentedData* fragmentedData, unsigned short i)
                {
                        return (*((unsigned short*) (fragmentedData->data->bytes + 2))) < i;
                });
        if (iterator == this->dataBuffer.end()) this->dataBuffer.push_back(fragmentedData);
        else this->dataBuffer.insert(iterator, fragmentedData);
}
else
{
        delete fragmentedData;
        return;
}

if (difference == 0)
{
        this->expectedId = (this->expectedId + 1) & 0x3fff;

        while (!this->dataBuffer.empty())
        {
                if (!giveData()) break;
        }
}
```

Ze všeho nejdřív se vytvoří a pošle datagram, který potvrzuje doručení. Bez ohledu na to, zdali je datagram fragmentován či nikoli, je obalen strukturou, která má dvě hodnoty typu *bool* pro uchování informací ohledně fragmentování. Ty je potřeba

přečíst, uložit a v původních datech vynulovat, aby mohlo být správně přečteno id datagramu.

Dalším úkolem je vypočítat rozdíl id mezi očekávaným datagramem a tím, který přišel. Pro tento účel je využívána metoda *getDifference*, která tento rozdíl zjistí. Metoda byla vytvořena pro tento zdánlivě triviální úkol z toho důvodu, že jelikož povolujeme cyklické opakování id, potřebujeme získat správné id i v takových případech, kdy očekávaná hodnota je 16383, což je nejvyšší možná hodnota, a příchozí id má hodnotu 0. Správný výsledek není -16383, nýbrž je 1. Nebo v případech, kdy očekávaná hodnota je 9000 a příchozí id je 0. Otázkou je, zdali se jedná o tak moc opožděný datagram nebo naopak o datagram, který je tak napřed, že už se zahájilo další kolo s identifikačními čísly. Tato metoda tyto situace řeší tak, že vyberu variantu s menší vzdáleností, čili tu, kde jsou čísla blíž sobě.

Po zjištění tohoto rozdílu se vyhodnotí podmínka. Jestliže se jedná o datagram, který předběhl nebo je to přímo ten, na který čekáme, vyhledáme správnou pozici v setříděném poli *std::deque* a na ni jej uložíme. Pokud by se jednalo o datagram s menším id než které očekáváme, je to starý a již zpracovaný datagram a odesílatel jen nepřijal informaci o jeho doručení. Proto jej smažeme a algoritmus ukončíme.

Navíc jestliže se jedná o právě očekávaný datagram, posuneme id očekávaného datagramu a přitom se však stále držíme v povolené hranici. Cyklicky voláme metodu *giveData()*, která se bude volat, dokud nebude buffer přijatých datagramů prázdný nebo dokud metoda nevrátí *false*, což znamená, že uživateli nemohla být data předána, neboť některý datagram chybí. Pokud vrátí *true*, znamená to, že data byla předána uživateli a můžeme zkusit, zdali se podaří předat další část dat. Každé její zavolání může uživateli dát maximálně jeden datagram nebo skupinu fragmentů. Její realizace je následující:

```
bool isFragmentComplete = false;
auto iterator = this->dataBuffer.begin();
unsigned short j = *((unsigned short*) ((*iterator)->data->bytes + 2));

for (iterator; iterator != this->dataBuffer.end(); iterator++)
{
        if (*((unsigned short*) ((*iterator)->data->bytes + 2)) != j) return false;
        else if ((*iterator)->isThisLastFragment)
        {
                isFragmentComplete = true;
                break;
        }
        else j = (j + 1) & 0x3fff; // držíme j v povolených hodnotách
}
if (!isFragmentComplete) return false;

int difference = std::distance(this->dataBuffer.begin(), iterator);
Data* wholeData;
```

```
if (difference == 0)
{
        wholeData = (*iterator)->data;
        (*iterator)->data = NULL;
        delete *iterator;
        this->dataBuffer.erase(iterator);
}
else
{
        int length = difference * (Communication::DATAGRAM_SIZE - 4)
                + (*iterator)->data->length;
        wholeData = new Data(length);
        int offset = 4;
        for (int i = 0; i <= difference; i++)
        {
                memcpy(wholeData->bytes + offset, this->dataBuffer[i]->data->bytes + 4,
                        this->dataBuffer[i]->data->length - 4);
                offset += this->dataBuffer[i]->data->length - 4;
                delete this->dataBuffer[i];
        }
        this->dataBuffer.erase(this->dataBuffer.begin(), ++iterator);
}

ConfirmatoryDataParser confirmatoryDataParser(wholeData);
processReceivedData(&confirmatoryDataParser);
delete wholeData;
```

Nastavíme iterátor na první datagram v seřazeném poli. Poté z něj načteme jeho id. Dále v cyklu postupujeme přes následující datagramy a hledáme takový, který by byl označen jako poslední fragment. To může být klidně hned ten první. Algoritmus musíme ukončit v případě, že se posuneme na následující datagram, ale jeho id je vyšší než o jedničku, to znamená, že je mezi datagramy díra a nějaký tedy chybí.

Pokud opravdu nalezneme takový datagram, jenž je označen jako poslední fragment, vypočítáme, kolik datagramů leží mezi ním a tím počátečním. Následující podmínka slouží pro případ, že počáteční a koncový fragment je jeden a ten samý datagram a tedy se jedná o nefragmentovaný datagram, abychom se vyhnuli kopírovaní, které je ve větvi *else*, a zpracování nefragmentovaných datagramů zrychlili. Jen nastavíme ukazatel na data, kopírování v tomto případě není potřebné. Z dataBufferu tento prvek odstraníme.

Ve větvi *else* na základně dříve vypočítaného počtu datagramů zjistíme velikost výsledných dat a ta alokujeme. Poté již jen z jednotlivých fragmentů překopírováváme data do celku a přitom postupně fragmenty mažeme. Po zkopírování vyčistíme pole a vytvoříme strukturu, do které předáme výsledná spojená data, a tu předáme uživateli.

### 4.9.2 MassChannel

MassChannel je typ kanálu navrhnutý přímo pro odesílání souborů. Také je zde velmi důležité, tak jako u ConfirmatoryChannel, aby byla přijatá data konzistentní. Velkou změnou je ale způsob, kterým se toho dosahuje. ConfirmatoryChannel potvrzuje každý datagram, na rozdíl od něj MassChannel potvrzuje velké množství dat najednou.

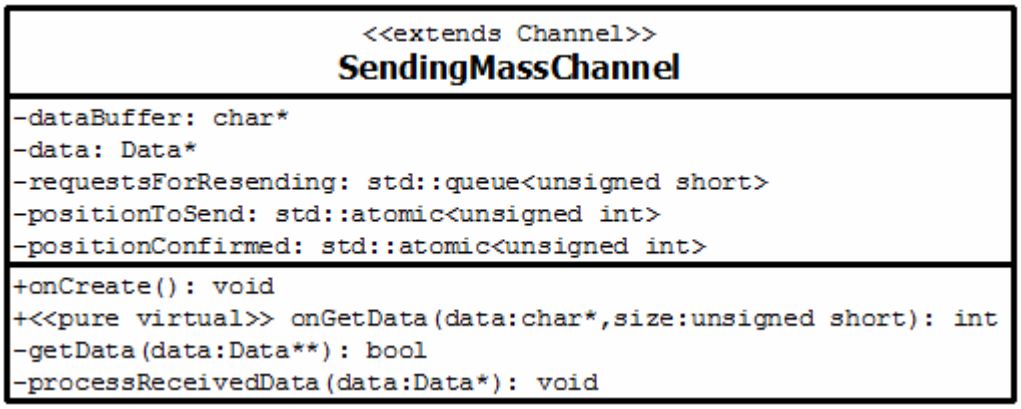
Potvrzení se tedy posílá jednou za čas v závislosti na výpadcích, obsahuje informace o chybějících datagramech a do jaké pozice je soubor přenesen.

Každý datagram přenášející uživatelova data obsahuje v rámci jednoho přenosu jedinečné identifikační číslo. První přenášený datagram má přiřazeno číslo 0. Maximální hodnota tohoto čísla je $2^{32}$, což dovoluje přenášet soubory o maximální velikosti $2^{32}$ * 1438 B, kde 1438 B je maximální velikost dat v datagramu. To je zhruba 5,6 TB.

Příjemce i odesílatel má k dispozici velký statický buffer (zhruba 20 MB). Příjemce ho využívá k řešení zaměněného pořadí přijatých datagramů, což u UDP může nastat, a také výpadků některých datagramů, které si po uplynutí jistého časového úseku znovu vyžádá. Proto má odesílatel buffer také k dispozici, aby mohl vyžádané datagramy znovu odeslat. Buffer se u něj maže až po doručení informace, že data byla skutečně protistranou přijata. Kdyby odesílatel dlouho nedostal informaci o přijetí dat, jeho buffer by se zcela naplnil, přenos by byl pozastaven a k pokračování by došlo až po přijetí takové informace. Příjemce zase buffer promaže až poté, co data pošle uživateli knihovny k zapsání do souboru.

**SendingMassChannel**

Implementování části pro odesílatele dat se provádí opět zděděním třídy *Channel* a překrytím dvou metod. Obrázek 10 popisuje strukturu této třídy.




Obrázek 10 Zjednodušený class diagram třídy SendingMassChannel

Realizace metody *processReceivedData(Data* data)*, která má za úkol zpracovávat informace o přijetí a nedoručených datagramech:

```
DataParser parser(data, 2);
unsigned int positionConfirmed = parser.getUint();

if (positionConfirmed < this->positionConfirmed) {}
```

```
else if (data->length == 7)
{
	unsigned int positionToSend = this->positionToSend;
	EnterCriticalSection(&this->CS_requestsForResending); // LOCK
	for (int j = positionConfirmed; j < positionToSend; j++) requestsForResending.push(j);
	LeaveCriticalSection(&this->CS_requestsForResending); // UNLOCK
	this->positionConfirmed = positionConfirmed;
	if (this->isEnd)
	{
		close();
	}
}
else if (positionConfirmed > this->positionConfirmed)
{
	this->positionConfirmed = positionConfirmed;

	short count = (data->length - 6) / 4;
	EnterCriticalSection(&this->CS_requestsForResending); // LOCK
	for (int j = 0; j < count; j++) requestsForResending.push(parser.getUint());
	LeaveCriticalSection(&this->CS_requestsForResending); // UNLOCK
	notifyDataReady();
}
else if (positionConfirmed == this->positionConfirmed)
{
	short count = (data->length - 6) / 4;
	EnterCriticalSection(&this->CS_requestsForResending); // LOCK
	if (requestsForResending.empty())
	{
		for (int j = 0; j < count; j++) requestsForResending.push(parser.getUint());
	}
	LeaveCriticalSection(&this->CS_requestsForResending); // UNLOCK
	notifyDataReady();
}

delete data;
```

Nejdříve si nastavíme *parser* dat, který posuneme na druhý byte, neboť nultý a první byte se používá k identifikaci kanálového čísla. Poté načteme číslo, které je v tomto potvrzovacím datagramu povinné, a které značí, že všechny datagramy s menším než tímto identifikačním číslem byly úspěšně přijaty. Díky tomu najednou potvrzujeme velký a kompletní blok dat. Stejné identifikační číslo si ve třídě uchováváme a vždy obsahuje maximální hodnotu ze všech přijatých potvrzení, takže postupem času, jak přenos plyne, narůstá.

Prvním podmínkou odfiltrujeme případy, kdy přijde staré potvrzení, neboť z toho, že třídní proměnná *this->positionConfirmed* je větší, je jasné, že v již v minulosti muselo přijít takové potvrzení, které obsahovalo vyšší identifikační číslo než které se v tomto potvrzovacím datagramu nachází.

Další podmínka se nejčastěji využívá na konci přenosu. Velikost datagramu 7 je speciální a říká, že příjemce už přijal datagram s jistým identifikačním číslem, ale žádný další, což může značit, že již přijal celý soubor. Tato informace se nachází v třídní proměnné *this->isEnd* a díky tomu se tento kanál uzavře, tím dojde ke korektnímu uzavření souboru i na druhé straně u příjemce.

Také je možné, že se ještě nejedná o konec souboru. Např. příjemci chybí jeden poslední datagram, který nebyl doručen, tedy jeho poslední datagram je vlastně předposlední. Pro tento případ se v tělo podmínky nachází cyklus *for*, který všechny zbývající datagramy přidal do fronty požadavků, aby mohly být znovu odeslány. Je vhodné dodat, že třídní proměnná *this->positionToSend* udává pozici posledního odesláno datagramu, mimo přeposlaných datagramů.

V následující podmínce testujeme, zdali má právě přijaté potvrzení vyšší identifikační číslo než předchozí potvrzení. To si vzápětí zaktualizujeme a poté na základě velikosti datagramu vypočítáme, kolik identifikačních čísel v sobě nese. Tato čísla udávají identifikační čísla nepřijatých datagramů. Jak vlastně příjemce zjistí, které datagramy mu nebyly doručeny? Pokud příjemce příjme datagram s identifikačním číslem např. 50 a před tím přijal všechny datagramy do 40, snadno vypočítá, které datagramy mu chybí.

V cyklu se přidají všechny nahlášené datagramy do fronty, aby mohly být přeposlány. Metoda *notifyDataReady* řeší jen tu situaci, kdy byl přenos kvůli přeplnění bufferu pozastaven a nyní má pokračovat.

Poslední podmínka řeší případ, kdy byla data přeposlána, ale opět nebyla přijata, tedy nemohlo dojít k navýšení proměnné *positionConfirmed*.

Druhou metodu, kterou jsme museli překrýt, je metoda *getData(Data** data)* vracející *bool*, kterou knihovna volá a žádá naplnění dat posílaných v argumentu:

```
bool isEmpty;
EnterCriticalSection(&this->CS_requestsForResending); // LOCK
isEmpty = this->requestsForResending.empty();
LeaveCriticalSection(&this->CS_requestsForResending); // UNLOCK

if (!isEmpty)
{
        unsigned int requestForResending;
        EnterCriticalSection(&this->CS_requestsForResending); // LOCK
        requestForResending = this->requestsForResending.front();
        this->requestsForResending.pop();
        LeaveCriticalSection(&this->CS_requestsForResending); // UNLOCK

        *data = &this->data[requestForResending % 15000];
        return true;
}

if (this->positionToSend - this->positionConfirmed >= 15000) return false;

unsigned int positionToSend = this->positionToSend;
Data& d = this->data[positionToSend % 15000];
int length = onGetData(d.bytes + 6, 1444 - 6);
if (length == 0)
{
        this->isEnd = true;
        return false;
}

d.length = length + 6;
```

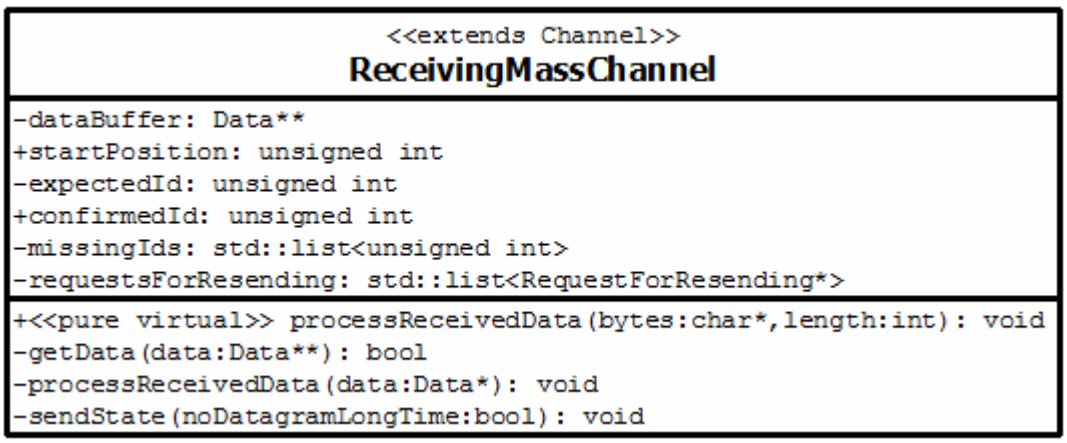
```
d.bytes[0] = getDestinationChannelNumber();
d.bytes[1] = getDestinationChannelNumber() >> 8;
d.bytes[2] = positionToSend;
d.bytes[3] = positionToSend >> 8;
d.bytes[4] = positionToSend >> 16;
d.bytes[5] = positionToSend >> 24;
this->positionToSend++;

*data = &d;

return true;
```

V kritické sekci ověříme, zdali je fronta požadavků na přeposlání některých datagramů prázdná. Pokud není, tak využijeme *buffer* a necháme odeslat první datagram s odpovídajícím identifikačním číslem nacházející se v této frontě. Poté ho z fronty odstraníme.

V případě, že je fronta prázdná, ověříme si, zdali můžeme pokračovat v odesílání nových datagramů. To provedeme odečtením proměnné *this->positionConfirmed*, která udává poslední nahlášení přijatých datagramů od příjemce, od proměnné *this->positionToSend*, která udává, u kterého čísla jsme v odesílání. Pokud je jejich rozdíl větší nebo roven než 15000, přenos je pozastaven. Ve většině případů však podmínka nevyhoví a algoritmus pokračuje dál. Od uživatele získá data, která uživatel rovnou naplní do *bufferu*, a ověří, jestli nemají nulovou délku. Pokud tomu tak je, značí to konec souboru a proto se proměnná *this->isEnd* nastaví na hodnotu *true*. V opačném případě se datům nastaví číslo kanálu a jeho identifikační číslo, inkrementuje se proměnná *this->positionToSend* a data se pomocí argumentu metody pošlou knihovně na odeslání.

### ReceivingMassChannel

Obrázek 11 zobrazuje strukturu této třídy.




Obrázek 11 Zjednodušený diagram třídy ReceivingMassChannel

Zpracování přijatých datagramů provádí překrytá metoda *processReceivedData(Data* data)*:

```
this->commingOfLastDatagramTime = Communication::timeInMiliseconds();

EnterCriticalSection(&this->CS_sendState);
unsigned int id = *((int*) &data->bytes[2]);

if (id == this->expectedId)
{
	if (this->missingIds.empty())
	{
		processReceivedData(data->bytes + 6, data->length - 6);
		delete data;
	}
	else
	{
		this->dataBuffer[
			(this->startPosition + id - *(this->missingIds.begin()) - 1) % 15000]
			= data;
	}
	this->expectedId++;
}
else if (id > this->expectedId)
{
	unsigned int position;
	if (this->missingIds.empty()) position = id - this->expectedId - 1;
	else position = (this->startPosition + id - *(this->missingIds.begin()) - 1) % 15000;
	this->dataBuffer[position] = data;

	for (unsigned int j = this->expectedId; j < id; j++) this->missingIds.push_back(j);
	this->expectedId = id + 1;
}
else if (id < this->expectedId)
{
	bool found = false;
	for (auto iterator = this->missingIds.begin(); iterator != this->missingIds.end();
		iterator++)
	{
		if (*iterator == id)
		{
			found = true;
			if (iterator == this->missingIds.begin())
			{
				processReceivedData(data->bytes + 6, data->length - 6);
				delete data;
				unsigned int count;
				unsigned int lastStartPosition = this->startPosition;
				if (this->missingIds.size() > 1)
				{
					iterator++;
					count = *iterator - id - 1;
					this->startPosition = (this->startPosition
						+ *iterator - id) % 15000;
				}
				else
				{
					count = this->expectedId - id - 1;
					this->startPosition = 0;
				}
				count += lastStartPosition;
				for (int j = lastStartPosition; j < count; j++)
				{
					processReceivedData(this->dataBuffer[j % 15000]->bytes
						+ 6, this->dataBuffer[j % 15000]->length - 6);
					delete this->dataBuffer[j % 15000];
				}
				this->missingIds.erase(this->missingIds.begin());
			}
			else
			{
```

```
                                this->dataBuffer[
                                        (this->startPosition + id - *(this->missingIds.begin())
                                        - 1) % 15000] = data;
                                this->missingIds.erase(iterator);
                        }
                        break;
                }
        }
        if (!found) delete data;
}

/*
  Zašleme potvrzení o přijatých datagramech, případně pošleme id datagramů,
  které nedorazily.
*/
if (this->expectedId - this->confirmedId > 3750)
{
        sendState(false);
}

LeaveCriticalSection(&this->CS_sendState);
```

Na začátku metody se podíváme na čas příchodu nejnovějšího datagramu. Tento čas potřebuje vlákno, které je nad tímto kanálem spuštěno, ale pouze u příjemce, k identifikaci přerušení datového toku, což může u odesílatele znamenat konec souboru nebo přeplněný *buffer*.

Celý zbytek kódu probíhá v kritické sekci, ve které se nejdříve z dat zjistí identifikační číslo datagramu (dále jen id). To se porovná s proměnnou *this->expectedId* (to je nejvyšší id, které dosud dorazilo, plus jedna, neboli jaké následující id se očekává) a kód se člení podle výsledku tohoto porovnání.

V ideálním případě se id rovná tomu, co se očekává, což je první možnost. V té se ještě ověřuje, zdali nebylo nějaké id přeskočeno a tedy nějaký datagram nechybí. V ideálním případě přicházejí datagramy v řadě za sebou. Pokud je seznam *this->missingIds* prázdný, data se předají uživateli k zapsání do souboru a následně se smažou. V případě, že seznam není prázdný, musíme data zapsat na příslušné místo do bufferu. Úkolem tohoto kanálu je uživateli dávat data v pořadí, v jakém byla odeslána. Takže pokud nějaký datagram nedorazí, všechny s vyšším id se musejí ukládat do bufferu.

Další možnost pro id zabezpečuje situaci, kdy přijde datagram s vyšším id než jaké se očekává, což znamená, že nějaké datagramy předběhl. V závislosti na tom, jestli je seznam *this->missingIds* prázdný, se vypočítá pozice v bufferu, na kterou se datagram uloží. Po jeho následném uložení se do seznamu vloží všechna id, která se nacházejí mezi hodnotami očekávaného id a id příchozího datagramu. Na závěr se vypočte nová hodnota pro proměnnou *this->expectedId*.

Poslední možnost nastane, pokud je id datagramu menší než očekávané id. V takovém případě se nejdříve v seznamu *this->missingIds* vyhledá jeho pozice. Pokud

se jedná o první chybějící datagram, čili můžeme pokračovat ve spojité řadě, dáme data uživateli a poté je smažeme. Dále musíme ověřit, jestli následující prvky v bufferu nenavazují na právě přijatý datagram. Zjištění počtu takovýchto datagramů zajišťují následující řádky. Nakonec je tento počet uložen v proměnné *count*. Pokud chybí více datagramů, proměnná *this->startPosition*, která vyjadřuje index v bufferu, se posune na další „díru“. Nebo v případě, že chyběl jen tento datagram, se nastaví na nulu. Potom následuje cyklus, který uživateli dává všechny takové datagramy, které jsou po sobě jdoucí. Nakonec se ze seznamu chybějící id odstraní.

V případě, že se o první chybějící datagram nejednalo, datagram se jen uloží do bufferu a záznam v seznamu se smaže. A pokud se id v seznamu chybějících datagramů nenajde, je datagram smazán.

V poslední větvi zavoláme metodu *sendState* jen tehdy, pokud je rozdíl mezi *this->expectedId* (nejvyšším přijatým id) a *this->confirmedId* (naposledy odeslanou potvrzující hodnotou) velký, a to tak, že odesílatel by měl v tu dobu mít buffer zaplněný ze čtvrtiny.

### 4.9.3 StreamChannel

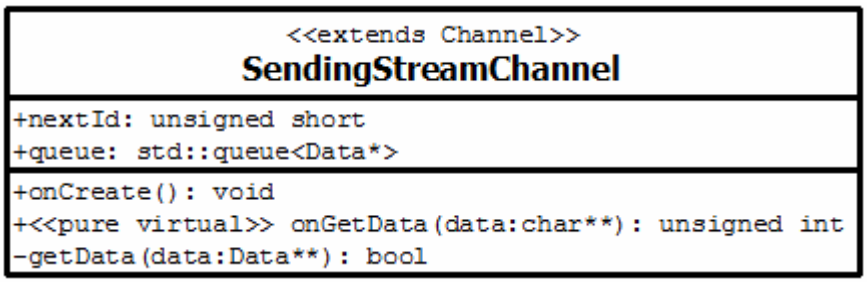
StreamChannel je typ kanálu určený pro streamování audia a videa. Jak jsem již několikrát naznačoval, přenos je nespolehlivý, což znamená, že příjemce neposílá odesílateli žádné potvrzovací datagramy. Jediné co příjemce řeší, je pořadí přijímaných datagramů. Když se stane, že datagram přeskočí nějaké jiné datagramy, tak se uživateli doručí ten přijatý a přeskočené datagramy se už nikdy nezpracují, a to ani tehdy, kdyby nakonec dorazily.

Podporována je jako u ConfirmatoryChannel také fragmentace datagramů, která funguje velmi podobně. Aby bylo možné fragmentovanou část dat zpracovat, musí dorazit všechny její fragmenty. Není možné předat uživateli jen část fragmentovaných dat, což se liší o nefragmentovaných dat, kde je přeskakování datagramů možné. Nicméně se samozřejmě jedná o negativní jev, který má zásadní vliv na kvalitu hovoru či přenosu videa.

#### SendingStreamChannel

Odesílání dat probíhá jinak než v případě ConfirmatoryChannel. O data žádá metoda *getData*, čili knihovna. To je z toho důvodu, že kdyby se data přidávala do bufferu ze strany uživatele, tak jak je tomu u ConfirmatoryChannel, a ta by se

nestíhala odesílat, buffer by postupně narůstal a narůstalo by zpoždění. Obrázek 12 znázorňuje strukturu této třídy.




Obrázek 12 Zjednodušený class diagram třídy SendingStreamChannel

Na straně odesílatele překrývá tento kanálový typ pouze jednu virtuální metodu *getData(Data** data)*:

```
if (!this->queue.empty())
{
	delete this->queue.front();
	this->queue.pop();
}

if (this->queue.empty())
{
	char* bytes;
	unsigned int size = onGetData(&bytes);
	if (size == 0)
	{
		notifyDataReady();
		return false;
	}

	unsigned short length;
	bool isThisFirstFragment = true;
	bool isThisLastFragment;
	unsigned int position = 0;
	Data* d;
	do
	{
		if (size - position > Communication::DATAGRAM_SIZE - 4)
		{
			length = Communication::DATAGRAM_SIZE - 4;
			isThisLastFragment = false;
		}
		else
		{
			length = size - position;
			isThisLastFragment = true;
		}
		d = new Data(length + 4);
		d->bytes[0] = getDestinationChannelNumber();
		d->bytes[1] = getDestinationChannelNumber() >> 8;
		d->bytes[2] = this->nextId;
		d->bytes[3] = (this->nextId >> 8) & 0x3f;
		this->nextId++;
		if (isThisFirstFragment) d->bytes[3] |= 0x80;
		if (isThisLastFragment) d->bytes[3] |= 0x40;
		isThisFirstFragment = false;
		memcpy(d->bytes + 4, bytes + position, length);
		position += length;
		this->queue.push(d);
	} while (!isThisLastFragment);
}
```

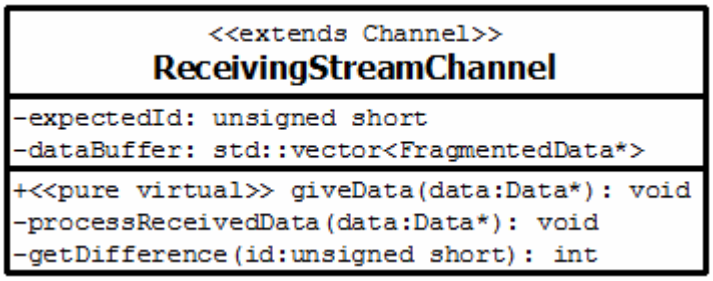
```
*data = this->queue.front();

return true;
```

V prvním kroku, když se knihovna „ptá“, zdali by tento kanál chtěl posílat data, se otestuje, jestli je fronta prázdná. Jestliže není, znamená to, že uživatel chtěl odesílat velká data, která musela být fragmentována. V těle první podmínky se vymaže první datagram i jeho zařazení do fronty *std::queue*. Jedná se o opožděný výmaz dat, která již byla v předchozím zavolání této metody odeslaná, ale není možné je smazat hned, jelikož se posílají až po vykonání této metody, takže bychom data vymazali dřív než by se odeslala. V další podmínce testuje totéž, neboť předchozí krok mohl stav změnit. Pokud je tedy fronta prázdná, znamená to, že všechna fragmentovaná data již byla odeslána. Pomocí virtuální metody *onGetData* od uživatele získáme další data. Zkontrolujeme, zdali jejich velikost není nulová, což by indikovalo, že zatím nejsou k dispozici žádná data. Ještě před ukončením se zavolá metoda *notifyDataReady*, která zabrání vláknu procházejícímu tento kód v uspání, jelikož se předpokládá, že tento typ kanálu potřebuje neustálé dotazování, zdali nechce posílat data.

V následujícím cyklu *do* se v případě potřeby data přijatá od uživatele rozfragmentují do datagramů. Jestliže jsou data dostatečně malá a vejdou se do jednoho datagramu, tak se vytvoří jen jeden datagram. Pro každý datagram je nastavena jeho hlavička, což zahrnuje nastavení id, údajů o fragmentaci a cílového kanálového čísla, a je do něj nakopírována část dat, aby byl posléze zcela připraven k odeslání. Díky tomuto cyklu se do fronty přidá minimálně jeden datagram. Po skončení cyklu se vybere k odeslání první datagram ve frontě.

**ReceivingStreamChannel**

Obrázek 13 ukazuje strukturu této třídy.




Obrázek 13 Zjednodušený class diagram třídy ReceivingStreamChannel

Příjem datagramů zajišťuje pouze jedna virtuální metoda, a to *processReceivedData(Data* data)*:

```
FragmentedData* fragmentedData = new FragmentedData;
fragmentedData->data = data;
fragmentedData->isThisFirstFragment = data->bytes[3] >> 7;
fragmentedData->isThisLastFragment = (data->bytes[3] & 0x7f) >> 6;
data->bytes[3] = data->bytes[3] & 0x3f;
unsigned short id = *((unsigned short*) (data->bytes + 2));

int difference = getDifference(id);
if (difference >= 0)
{
	auto iterator = std::lower_bound(this->dataBuffer.begin(), this->dataBuffer.end(), id,
		[](const FragmentedData* d, unsigned short j)
		{
			return (*((unsigned short*) (d->data->bytes + 2)) < j);
		});

	if (iterator == this->dataBuffer.end()) // fronta je prázdná
	{
		this->dataBuffer.push_back(fragmentedData);
		iterator = --this->dataBuffer.end();
	}
	else if (*((unsigned short*) ((*iterator)->data->bytes + 2))
		== *((unsigned short*) (data->bytes + 2))) // přišel duplicitní datagram
	{
		delete fragmentedData;
		return;
	}
	else iterator = this->dataBuffer.insert(iterator, fragmentedData);
	bool isFragmentComplete = false;
	unsigned short j = id;
	std::vector<FragmentedData*>::reverse_iterator startBoundRIt(iterator);
	startBoundRIt--;
	for (startBoundRIt; startBoundRIt != this->dataBuffer.rend(); startBoundRIt++)
	{
		if (*((unsigned short*) ((*startBoundRIt)->data->bytes + 2)) != j) return
		else if ((*startBoundRIt)->isThisFirstFragment)
		{
			isFragmentComplete = true;
			break;
		}
		else j = (j - 1) & 0x3fff; // držíme j v povolených hodnotách
	}
	if (!isFragmentComplete) return;

	isFragmentComplete = false;
	j = id;
	std::vector<FragmentedData*>::iterator endBoundIt(iterator);
	for (endBoundIt; endBoundIt != this->dataBuffer.end(); endBoundIt++)
	{
		if (*((unsigned short*) ((*endBoundIt)->data->bytes + 2)) != j) return;
		else if ((*endBoundIt)->isThisLastFragment)
		{
			isFragmentComplete = true;
			break;
		}
		else j = (j + 1) & 0x3fff; // držíme j v povolených hodnotách
	}
	if (!isFragmentComplete) return;

	int difference = std::distance(--(startBoundRIt.base()), endBoundIt);
	int length = difference * (Communication::DATAGRAM_SIZE - 4)
		+ (*endBoundIt)->data->length - 4;
	Data* wholeData = new Data(length);
	int offset = 0;
	std::vector<FragmentedData*>::iterator it(--(startBoundRIt.base()));
	for (int i = 0; i <= difference; i++)
	{
		memcpy(wholeData->bytes + offset, (*it)->data->bytes + 4,
			(*it)->data->length - 4);
		offset += (*it)->data->length - 4;
		it++;
	}
	this->expectedId = (*((unsigned short*) ((*(--it))->data->bytes + 2)) + 1) & 0x3fff;
```

```
        it++;
        for (auto dIt = this->dataBuffer.begin(); dIt != it; dIt++)
        {
                delete (*dIt);
        }
        this->dataBuffer.erase(this->dataBuffer.begin(), it);

        giveData(wholeData);
}
else delete fragmentedData;
```

Začátek kódu je stejný jako u ReceivedConfirmatoryChannel. Po zjištění rozdílu metodou *getDifference*, která je také stejná jako u ConfirmatoryChannel, už je kód odlišný. Pokud je rozdíl id příchozího datagram a id očekávaného datagramu nulový nebo kladný, vyhledá se místo v bufferu pro uložení datagramu tak, aby i po jeho vložení zůstal buffer seřazený dle id. Jestliže po vyhledání zjistíme, že datagram se shodným id už je v bufferu uložen, nově přijatý vymažeme a metodu ukončíme. V ostatních případech na vyhledané místo přijatý datagram uložíme a pokračujeme dál. Danou pozici si zapamatujeme, neboť z ní budeme vycházet.

V prvním cyklu jdeme v datech směrem doleva a snažíme se najít datagram, který je označený jako první fragmentovaný, kdy stále kontrolujeme, zdali je id datagramu vždy o jedničku menší než id toho předchozího. V dalším cyklu stejným způsobem hledáme naopak poslední fragmentovaný datagram.

Nyní jsme ve stavu, kdy známe pozici počátečního i koncového datagramu fragmentovaných dat. Vypočítáme si tedy vzdálenost, která je mezi nimi. Z tohoto údaje spočítáme délku složených dat, která z těchto fragmentů vzniknou, a místo pro ně alokujeme. Poté do těchto dat postupně nakopírujeme z jednotlivých fragmentů jejich části dat. Pokračujeme vymazáním nejen zpracovaných dat, ale také všech těch, která jsou před nimi, protože v danou chvíli se jedná o stará a nepotřebná data, která stejně dosud nejsou kompletní. Také se odstraní ukazatele z bufferu. Nakonec se uživateli předají zkompletovaná data.

Kdyby se stalo, že by došel datagram, jehož id je menší než očekávané, byl by vymazán a algoritmus by skončil.

## 4.10 Optimalizace rychlosti spojení

Jelikož jsme nuceni používat protokol UDP, musíme se zabývat i tím, jakou rychlostí data odesílat. TCP má mechanismus, který toto zajišťuje automaticky.[8] UDP ničím takovým nedisponuje, a je potřeba to řešit na vyšší úrovni, tedy v aplikaci. Většinou jsme omezeni poskytovatelem internetového připojení. Reálnou rychlost internetového připojení však počítač nezná.

Jediným řešením je postupně rychlost zvyšovat a neustále sledovat, zdali můžeme v tomto zvyšování pokračovat. Jenže co budeme sledovat, abychom dostali požadovanou odpověď? Dají se sledovat dva údaje. Jestliže budeme pořád zvyšovat rychlost, jistě se dostaneme na takovou úroveň, kdy už nebude router stíhat datagramy odesílat, a jeho buffer se začne pomalu zaplňovat. Datagramy budou na routeru uloženy ve frontě a budou čekat na odeslání stále déle. Díky tomuto menšímu zahlcení se jistě začne zvyšovat latence a tím zjistíme, že jsme narazili na rychlostní limit. K výpočtu latence slouží tzv. REFRESHING datagramy, které se odesílají dostatečně často, aby údaj o momentální latenci byl co možná nejaktuálnější.

Druhým údajem, který můžeme sledovat, jsou výpadky datagramů. Problém tohoto sledování je v tom, že datagramy mohou vypadávat i za normální situace, což by se dalo odfiltrovat prahovou hodnotou. Největší problém je v tom, že obrovský výpadek datagramů nastane až v případě, že se datagramy nevejdou do bufferu na routeru a ten je začne zahazovat. To už je stav, ke kterému by dojít vůbec nemělo, a který může způsobit až restartování routeru, což se mi při testování stalo. S tímto přístupem sledování by každý kanál musel optimalizovat rychlost svého odesílání dat, a k tomu účelu by pro každý kanál muselo být spuštěno vlákno. Přitom stačí optimalizovat rychlost pro celé spojení, tedy pro všechny kanály jednoho spojení najednou, protože tyto kanály by se chovaly stejně, a tudíž by bylo zbytečné řídit rychlost u každého zvlášť. Proto využijeme pouze latenci.

### 4.10.1 Získání hodnoty latence

Nejdříve si musíme zajistit aktuální údaj o latenci daného spojení, abychom jej poté mohli využít pro výpočet rychlosti. Odesílání REFRESHING datagramů zajišťuje třída *Communication* nám již známou metodou *runRefreshing(char* data)* pro všechna spojení najednou. Každému spojení je zavolána metoda *refresh*, jejíž zdrojový kód je následující:

```
Refreshing* refreshing = new Refreshing;
refreshing->id = this->nextSentId;
refreshing->time = Communication::timeInMiliseconds();

time_t firstPendingRefresh;

EnterCriticalSection(&this->CS_listOfRefreshing);
this->listOfRefreshing.push_back(refreshing);
firstPendingRefresh = (*this->listOfRefreshing.begin())->time;
LeaveCriticalSection(&this->CS_listOfRefreshing);

if (Communication::timeInMiliseconds() - firstPendingRefresh > Communication::INACTIVE_TIME)
        disconnect();
```

```
data[3] = this->nextSentId;
data[4] = this->nextSentId >> 8;
this->nextSentId++;
this->communication->send(this->destinationAddress, data, 5);
```

Vytvoříme si strukturu, do které uložíme id a aktuální čas. V kritické sekci tuto vytvořenou strukturu uložíme do seznamu a ještě si načteme čas odeslání prvního nepřijatého REFRESHING datagramu. Pokud je rozdíl tohoto času s aktuálním časem větší než stanovená hodnota, spojení bude přerušeno. Jinak se do datagramu uloží na správnou pozici jeho id a odešle se.

Zpracování tohoto typu řídící informace je velmi snadné. V přijatém datagramu se změní typ z REFRESHING na REFRESHING_CONFIRMATION (jeho id zůstává nezměněno) a pošle se zpátky.

Příjem REFRESHING_CONFIRMATION (zkráceně RC) datagramu zajišťuje metoda *processRefresh(unsigned short id)* třídy *Connection*, čili každé spojení si tuto informaci zpracovává samo:

```
if ((id > this->lastReceivedId || this->lastReceivedId - id > 30000)
        && (id <= this->nextSentId || id - this->nextSentId > 30000))
{
        int length = id - this->lastReceivedId - 1;
        if (length < 0) length += USHRT_MAX + 1;
        this->lastReceivedId = id;

        EnterCriticalSection(&this->CS_listOfRefreshing); // LOCK
        auto iterator = this->listOfRefreshing.begin();
        for (int j = 0; j < length; j++)
        { // Odstraníme všechny starší refreshe, ty už nás stejně nezajímají
                delete (*iterator);
                iterator = this->listOfRefreshing.erase(iterator);
        }

        time_t time = (*iterator)->time;
        delete (*iterator);
        this->listOfRefreshing.erase(iterator);
        LeaveCriticalSection(&this->CS_listOfRefreshing); // UNLOCK

        time_t currentTime = Communication::timeInMiliseconds();
        unsigned short latency = currentTime - time;

        if (this->localMinLatency > latency) this->localMinLatency = latency;
        if (this->globalMinLatency > latency) this->globalMinLatency = latency;

        EnterCriticalSection(&this->CS_latency); // LOCK
        this->latency[this->pointerToLatency] = latency;
        this->pointerToLatency = (++this->pointerToLatency) % 3;
        recountSpeed();
        LeaveCriticalSection(&this->CS_latency); // UNLOCK
}
```

Zpracovávat přijatý RC datagram budeme jen v tom případě, že jeho id je větší než jaké měl předchozí RC datagram. Podmínka myslí i na přetečení čítače. Pokud je splněna, vypočítá se rozdíl mezi id přijatého RC datagramu a id naposledy přijatého RC datagramu. Kdyby byl rozdíl záporný, přičte se k němu jedna perioda. To řeší případ,

kdy poslední přijaté id je těsně před přetečením a přijaté id již přeteklo, takže jejich rozdíl by byl záporný. Poté aktualizujeme id posledního přijatého RC datagramu.

V kritické sekci odstraníme všechny záznamy starších id. Načteme si čas odeslání REFRESHING datagramu se shodným id, a poté prvek ze seznamu odstraníme.

Následně vypočítáme rozdíl aktuálního času s časem odeslání, tedy jak dlouho trvalo datagramu dojít k protistraně a zase zpátky, což je latence. Otestujeme, zdali tato latence není nejnižší, pokud je, hodnotu aktualizujeme.

V jiné kritické sekci uložíme latenci do pole, ve kterém uchováváme poslední tři výpočty latence, a zavoláme metodu *recountSpeed* na přepočet rychlosti.

### 4.10.2 Výpočet rychlosti

Díky tomu, že k řízení rychlosti využíváme pouze latenci, je její řízení velmi náročné. REFRESHING datagramy se mohou ztrácet nebo zpožďovat i v případech, kdy v síti není žádný problém. Vycházíme tedy z toho, že tento údaj je zatížen šumem, a ten je nutné odfiltrovat.

Základem řízení maximální rychlosti je pole tří posledních hodnot latencí. Při každém přepočtu se z tohoto pole vyberou právě dvě hodnoty. Jako první hodnota je vybráno minimum z prvního a druhého prvku pole. Druhá hodnota je minimum druhého a třetího prvku pole. Kdyby se v poli vyskytla jedna hodnota, která by byla výrazně vyšší než ty ostatní, pravděpodobně by se jednalo jen o šum, a díky tomuto mechanismu výběru by se tato informace zatížená šumem vůbec nebrala v úvahu. Abychom problém začali detekovat, muselo by zvýšení latence postihnout dva prvky pole.

Podívejme se na metodu *recountSpeed*:

```
this->newestLatency = min(this->latency[(this->pointerToLatency + 1) % 3],
        this->latency[(this->pointerToLatency + 2) % 3]);
unsigned short olderLatency = min(this->latency[this->pointerToLatency],
        this->latency[(this->pointerToLatency + 1) % 3]);

if (this->isProblem)
{
        if (this->newestLatency < 3 * this->globalMinLatency
                || Communication::timeInMiliseconds() - this->timeOfProblem >= MAX_TIME_TO_WAIT)
                this->isProblem = false;
}
else if (this->newestLatency > 7 * this->globalMinLatency) // Mechanismus poslední záchrany
{
        this->speed = MIN_SPEED;
        this->timeOfProblem = Communication::timeInMiliseconds();
        this->isProblem = true;
}
else if ((this->newestLatency + 10) > 1.8 * (this->localMinLatency + 10))// Mechanismus záchrany
{
        this->speed = this->basicSpeed / 2;
        this->localMinLatency = this->newestLatency;
```

```
}
else
{
        float ratio = (float) (olderLatency + 10) / (this->newestLatency + 10);
        unsigned int real = getAndRecountRealSpeed();

        if (ratio < UPPER_LIMIT_RATIO_FOR_SLOWDOWN)
        {
                // zpomalení
                this->speed = this->basicSpeed * ratio;
                if (this->speed < 2000) this->speed = 2000;
        }
        else
        {
                if ((float) real / (this->speed + 1) < 0.5F) return;

                // zrychlení
                unsigned int speed = this->basicSpeed * SPEED_MULTIPLIER_FOR_SPEEDUP;
                if (speed < this->speed) this->speed = UINT32_MAX;
                else this->speed = speed;
        }
}
this->basicSpeed = this->speed;
```

Provedeme tedy výběr dvou hodnot z pole tří posledních hodnot latencí. Následuje sekvence podmínek.

První podmínka vyhoví, pokud byl detekován skutečně vážný problém, neboť dokud je proměnná *this->isProblem* nastavená na *true*, spojení neodešle jediný datagram. Nastavena na *false* může být, až se latence sníží na méně než trojnásobek nejnižší latence za celou dobu trvání spojení nebo po uplynutí stanoveného času.

Druhá podmínka nastavuje proměnnou *this->isProblem* v případě, kdy je latence vyšší než sedminásobek nejnižší latence. Navíc je rychlost nastavena na počáteční (nízkou) hodnotu, aby se poté začalo odesílat nejnižší rychlostí, a ne tou, která způsobila tento „krizový“ stav.

Třetí podmínka je už mírnější. Pokud je rychlost vyšší než daný násobek lokálního minima latence, rychlost se sníží na polovinu a lokální minimum se nastaví na nejnovější hodnotu latence. Proč nastavujeme lokální minimum na hodnotu, která je vysoká? Dejme tomu, že na začátku spojení máme latence kolem 50 ms. Tato hodnota je uložena jak v lokálním, tak v globálním minimu. Jenže poté uživatel začne stahovat soubor z Internetu a díky tomu se latence zvýší a bude kolísat mezi 100 ms až 200 ms. Změnily se podmínky, takže se jim musíme přizpůsobit, jinak by se rychlost snižovala až na nejnižší hodnotu.

V jiném případě si vypočítáme poměr starší latence s tou novější. Dále si přepočteme skutečnou rychlost odesílání (dále bude vysvětleno). Pokud je nejnovější latence vyšší než ta starší, rychlost se tímto poměrem vynásobí, tím se rychlost sníží o velikost rozdílu mezi latencemi. Pokud je naopak nejnovější latence zhruba stejná jako ta starší, rychlost se vynásobí konstantou, aby se zvětšila. Ještě před tím se ale

otestuje, zdali vůbec nějaká data odesíláme, protože rychlost, kterou optimalizujeme, je pouze horní limit rychlosti, kterou můžeme data posílat. Kdybychom nic neodesílali, horní limit rychlosti by stále rostl, latence by byla stále v pořádku, a jakmile bychom začali odesílat, data by se odesílala obrovskou rychlostí a rychle by došlo k zahlcení. Jelikož nepotřebujeme zvyšovat maximální limit rychlosti, pokud toho současného limitu nedosahujeme, ukončíme v této fázi tuto metodu.

V případě, že se skutečná rychlost odesílání blíží horní hranici, a je potřeba tuto hranici posunout výše, zvýšení rychlosti uložíme.

### 4.10.3 Odesílání datagramů

Odesílání provádí vlákno, které má vytvořeno každé spojení. Zdrojový kód vlákna je následující:

```
Data* data = NULL;
float preciseWait = 0;
int wait = 0;
time_t time;
time_t sleepTime;
time_t sleepFor = 0;
unsigned short latency;
unsigned short difference;

while (!this->isInterrupted)
{
	if (!Connection::getData(&data))
	{
		EnterCriticalSection(&this->CS_sending); // LOCK
		if (!this->areDataAlreadyReady)
		{
			sleepTime = Communication::timeInMiliseconds();
			SleepConditionVariableCS(&this->CV_sending, &this->CS_sending,
				INFINITE);
			sleepFor = Communication::timeInMiliseconds() - sleepTime;
		}
		this->areDataAlreadyReady = false;
		LeaveCriticalSection(&this->CS_sending); // UNLOCK

		if (sleepFor > 100)
		{
			EnterCriticalSection(&this->CS_latency); // LOCK
			this->basicSpeed = 2 * this->basicSpeed / log10(sleepFor);
			if (this->basicSpeed < MIN_SPEED) this->basicSpeed = MIN_SPEED;
			this->speed = this->basicSpeed;
			LeaveCriticalSection(&this->CS_latency); // UNLOCK
			sleepFor = 0;
		}
	}
	else
	{
		this->communication->send(this->destinationAddress, data->bytes, data->length);

		EnterCriticalSection(&this->CS_realSpeed); // LOCK
		this->lengthSum += data->length;
		LeaveCriticalSection(&this->CS_realSpeed); // UNLOCK

		if (getAndRecountRealSpeed() / (this->speed + 1) >= 0.5F)
		{
			difference = this->nextSentId - this->lastReceivedId - 1;
			if (difference < 0) difference += USHRT_MAX + 1;
			if (difference > 0)
```

```
                        {
                                EnterCriticalSection(&this->CS_listOfRefreshing);
                                if (this->listOfRefreshing.empty())
                                        time = Communication::timeInMiliseconds();
                                else time = (*this->listOfRefreshing.begin())->time;
                                LeaveCriticalSection(&this->CS_listOfRefreshing);

                                latency = Communication::timeInMiliseconds() - time;

                                EnterCriticalSection(&this->CS_latency); // LOCK
                                float ratio = (float) (this->newestLatency + 10)
                                        / (latency + 10);
                                if (ratio < UPPER_LIMIT_RATIO_FOR_SLOWDOWN)
                                {
                                        this->speed = this->basicSpeed * ratio;
                                        if (this->speed < 2000) this->speed = 2000;
                                }
                                LeaveCriticalSection(&this->CS_latency); // UNLOCK
                        }
                }

                preciseWait += 1000 * data->length / this->speed;
                wait = preciseWait;
                if (wait >= 2)
                {
                        preciseWait -= wait;
                        EnterCriticalSection(&this->CS_pausing); // LOCK
                        SleepConditionVariableCS(&this->CV_pausing, &this->CS_pausing, wait);
                        LeaveCriticalSection(&this->CS_pausing); // UNLOCK
                }

                while (this->isProblem)
                {
                        EnterCriticalSection(&this->CS_pausing); // LOCK
                        SleepConditionVariableCS(&this->CV_pausing, &this->CS_pausing, 100);
                        LeaveCriticalSection(&this->CS_pausing); // UNLOCK
                }

                EnterCriticalSection(&this->CS_sending); // LOCK
                this->areDataAlreadyReady = false;
                LeaveCriticalSection(&this->CS_sending); // UNLOCK
        }
        data = NULL;
}
```

Nejdříve se otestuje, zdali nějaký kanál nechce odesílat data. Jestliže nechce, vstoupíme do kritické sekce, ve které se ještě otestuje, zdali se to mezitím nezměnilo. Pokud ne, uložíme si čas a vlákno uspíme. To může být opět vzbuzeno jen některým z kanálů. Opustíme kritickou sekci, a pokud bylo vlákno uspáno déle než 100 ms, snížíme rychlost, protože se mezitím mohl změnit stav v síti.

V případě, že některé z vláken chce odesílat data, jdeme do větve *else*. Data se fyzicky odešlou. Do proměnné *this->lengthSum* přičteme velikost odeslaného datagramu. To slouží k výpočtu skutečné rychlosti odesílání.

Následuje velmi důležitá část, která v případě, kdy je skutečná rychlost odesílání větší než polovina horní hranice rychlosti, řeší situaci, kdy se delší dobu nevrací REFRESHING datagramy. To může být způsobeno buď výpadkem REFRESHING datagramu nebo zahlcením sítě. Dále se testuje, jestli je rozdíl mezi posledním odeslaným id a posledním přijatým id větší než 0, čili odpověď na nějaký datagram

nedorazila. Pokud ještě stihl datagram dorazit před vstupem do kritické sekce, je čas nastaven na současný čas a algoritmus se sice provede, avšak nebude mít na změnu rychlosti vliv. Nejpravděpodobněji to za tak krátkou chvíli nestihne a je čas nastaven na čas odeslání prvního odeslaného a zároveň nepřijatého REFRESHING datagramu. Poté vypočítáme latenci a snížíme rychlost dle vypočítaného poměru současné latence a nejnovější latence vypočítané předchozí metodou. Proto máme dvě proměnné *this->speed* a *this->basicSpeed*. První z nich se využívá k doladění rychlosti tímto mechanismem. Druhou z nich nastavuje jen předchozí metoda.

Z velikosti odeslaného datagramu a aktuální rychlosti vypočteme, na jak dlouho by se mělo odesílající vlákno uspat. Tento čas se postupně sčítá a až je větší než 2 ms, skutečně se vlákno uspí.

Nakonec pokud je síť zahlcena, přeruší se odesílání, a vlákno je uspáno. Pokračovat může až po přenastavení proměnné *this->isProblem* na *false*.

Na závěr se ještě podíváme na výpočet skutečné rychlosti odesílání, který provádí metoda *getAndRecountRealSpeed*. Ta se počítá jako průměrná rychlost za časový úsek větší nebo roven 100 ms. Uživateli je navrácena buď dříve vypočítaná skutečná rychlost v případě, že metoda byla zavolána znovu za méně než 100 ms, nebo ta nově vypočtená hodnota.

## 4.11 Implementace knihovny pro podporu sítě

Pro implementování knihovny stačí zdědit třídu *Communication*, kde je nutné překrýt několik virtuálních metod. Tou nejdůležitější je metoda *processCreatingConnection*, ve které je potřeba vytvářet svou třídu zděděnou ze třídy *Connection*. Tuto metodu knihovna zavolá v případě, že obdrží žádost o vytvoření spojení nebo potvrzení žádosti o spojení.

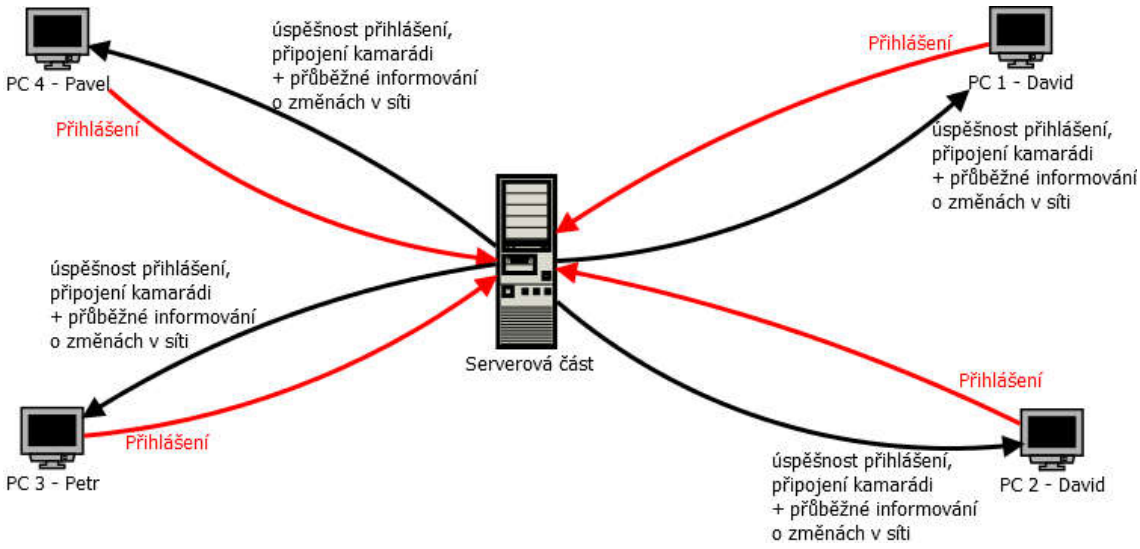
Dále je nutné překrýt metodu *processDisconnection*, pomocí které knihovna jejího uživatele informuje o ukončení spojení. Poslední metodou, kterou je nutné překrýt, je metoda *unableToConnect*, která uživatele informuje o skutečnosti, že se nepovedlo spojit s jinou stanicí.

Další třídou, kterou je nutné zdědit, je třída *Connection*. V ní se opět nachází několik virtuálních metod. První z nich je metoda *onCreate*, která je volána hned poté, co je spojení korektně vytvořeno. Toho se dá využít pro tvorbu komunikačního kanálu nebo pro oznámení, že spojení bylo vytvořeno.

Další virtuální metodou je metoda *creatingChannel*. Její překrytí umožní vytvořit další typy kanálů, což se hodí jen v tom případě, kdy nám ty základní nestačí. Dále můžeme překrýt skupinu metod s názvem *procesCreateChannel*, u kterých se mění první parametr podle typu kanálu. Pro každý typ jsou dvě metody, např. *SendingMassChannel* a *ReceivingMassChannel*. Stačí překrýt jen ty metody s kanálovými typy, jež budeme v dané aplikaci potřebovat. Tato skupina metod slouží k inicializaci tříd dědících od zvoleného kanálového typu. V další části implementace musíme vytvořit právě tyto třídy. Každý typ kanálu vyžaduje překrytí odlišných virtuálních metod. O tom, které metody je třeba překrýt u kterých kanálových typů, pojednává předchozí kapitola.

# 5 Architektura sítě

Každý klient se nejdříve musí připojit k serveru. Díky tomu se vytvoří spojení, které si klient bude držet. Po úspěšném přihlášení se nenavazuje spojení s jinými uživateli. To se provádí až po tom, co uživatel otevře konverzaci. Obrázek 14 znázorňuje situaci, kdy jsou klienti připojeni pouze k serveru a tedy mezi nimi zatím neexistuje žádné spojení. Z důvodu přehlednosti nejsou routery uživatelů na obrázku znázorněny.




Obrázek 14 Architektura sítě po přihlášení

Tato situace může trvat libovolný čas.

## 5.1 Selhání postupu v konkrétní situaci

Během testování jsem narazil na situaci, ve které UDP hole punching s využitím server nefungoval. Problém je, jestliže máme v LAN síti spuštěnou serverovou a zároveň klientskou aplikaci, a v jiné síti se na nachází druhý klient. Tito dva klienti spolu nemohou navázat spojení. Příčina může být v mém routeru, jelikož se nejedná o žádný profesionální router.

Pro správné fungování buď musí být serverová i klientské aplikace v jedné LAN síti nebo serverová aplikace musí být spuštěna na serveru s přidělenou veřejnou IP adresou nebo případně nacházejícím se v jiné LAN než všichni ostatní klienti.

# 6 Serverová aplikace

Serverová aplikace běží jako služba na pozadí na dedikovaném serveru. Mezi její činnosti patří:

- přihlašování uživatelů,
- informování přihlášených uživatelů o změnách,
- asistování během spojování dvou uživatelů.

## 6.1 Činnosti serverové aplikace

### 6.1.1 Přihlašování uživatelů

Poté co uživatel na svém počítači vyplní přihlašovací formulář, klientská aplikace (dále jen klient) nejdříve naváže spojení se serverem a obě strany otevřou komunikační kanál typu ConfirmatoryChannel. Následně klient odešle žádost o challenge[1]. Serverová aplikace (dále zkráceně server) jej tedy vygeneruje, uloží a klientovi odešle. Klient zahashuje heslo, které uživatel zadal, a hned poté jej zahashuje znovu spolu s přijatým challengem. Schématicky by se tato činnost dala vyjádřit takto: hash(hash(salt + heslo + uživatelské jméno) + challenge). Následuje odeslání přihlašovacího jména uživatele a jeho dvakrát zahashovaného hesla.

Server pak porovná toto dvakrát zahashované heslo s hashem hesla z databáze, které se ještě jednou zahashuje spolu s challengem, jenž si předtím server uložil. V databázi jsou uloženy hashe hesel vytvořené takto: hash(salt + heslo + uživatelské jméno). Server nakonec uživateli odešle výsledek přihlášení. Pokud byl přístup povolen, jsou spolu s výsledkem odeslány informace o připojených kontaktech, jejich zařízeních včetně IP adres i portů.

Tento poměrně komplikovaný způsob přihlášení se provádí z důvodu bezpečnosti. Jelikož (zatím) není komunikace šifrována, přenášení hesla v plain textu není dobrý nápad. Proto se provádí jeho hashování s přidanou solí. Jenomže kdyby se útočníkovi podařilo odposlechnout komunikaci, k přihlášení by nepotřeboval heslo v plain textu, stačilo by mu ono hashované heslo. Proto se používá challenge, díky němuž je odesílané heslo od uživatele při každém přihlášení jiné a funguje jen jednou. Takže i kdyby se útočníkovi heslo podařilo odposlechnout, tak už jej spolu

[1] Challenge je náhodně vygenerovaný řetězec znaků, který se přimíchává k heslu ve fázi hashování.

s uživatelským jménem nemůže stihnout poslat na server. Jakmile žádost legitimního uživatele přijde na server, challenge ztrácí platnost a útočníkovi je heslo k ničemu.

### 6.1.2 Informování přihlášených uživatelů o změnách

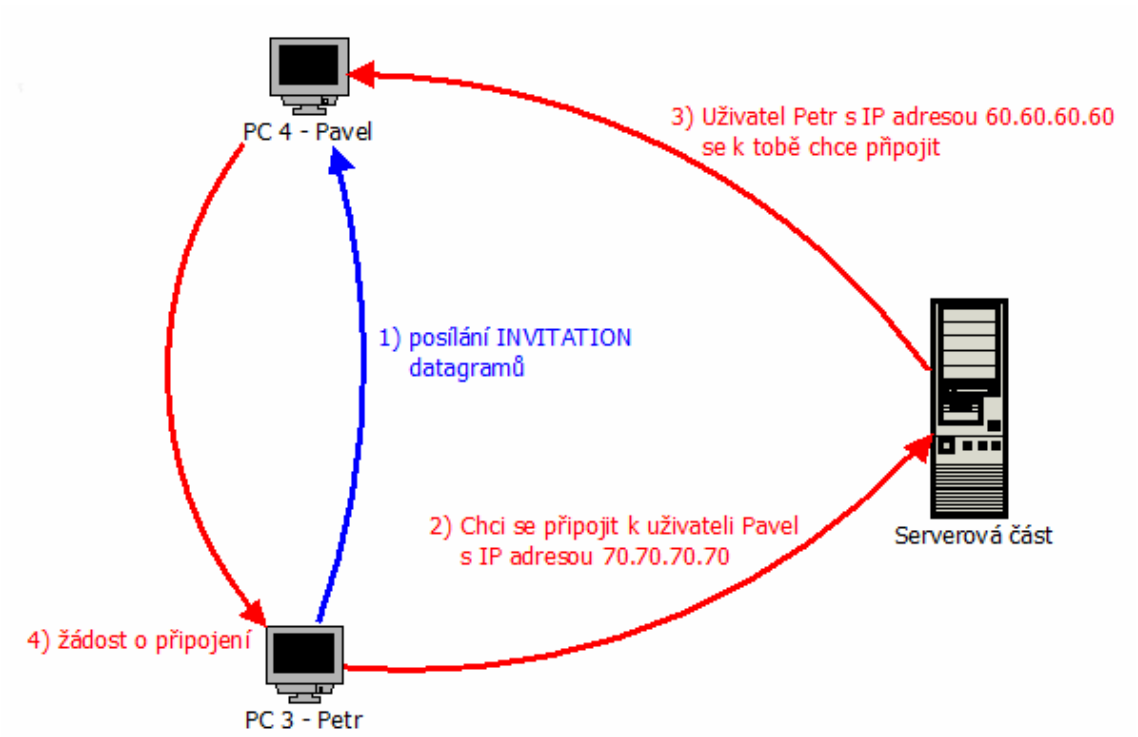
Jestliže se nějaký uživatel odpojí, server zjistí, že spojení s ním bylo přerušeno, a o této skutečnosti informuje všechny jeho kontakty. Stejně tak, když se uživatel nebo jeho další zařízení připojí, server opět informuje jeho kontakty. Spolu s touto informací se odesílá i jeho IP adresa a port.

Server má spojení se všemi uživateli po celou dobu jejich připojení, na rozdíl od klientů, kteří mezi sebou vytvářejí spojení jen vyžádání jednoho z nich.

### 6.1.3 Asistování během spojování dvou uživatelů

Tato činnost je vlastně činností STUN serveru, který v případě serverové aplikace umí i další věci.

Obrázek 15 popisuje situaci, kdy uživatel Petr bude chtít otevřít konverzaci s uživatelem Pavlem.



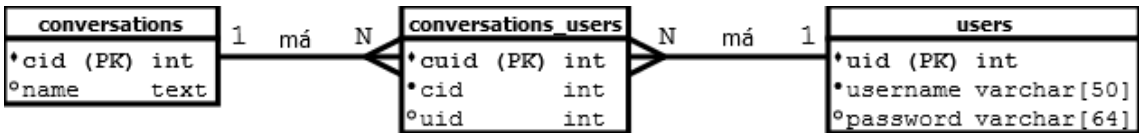

Obrázek 15 Proces spojení dvou uživatelů

Pokud se jeden uživatel rozhodne navázat spojení s některým kontaktem ve svém kontaktním listu, nejdříve začne v pravidelných intervalech posílat INVITATION[1] datagramy směrem k cílovému počítači a poté pošle na server žádost, že by rád navázal spojení se zařízením zvoleného uživatele. Server, který stále má se svými klienty navázané spojení, přepošle tuto žádost cílovému uživateli. Ten se pokusí na žádajícího klienta připojit.

## 6.2 Struktura databáze

Na následujícím obrázku vidíme jednoduchou, ale dostačující strukturu databáze. Všechny ostatní informace o klientech si uchovává přímo serverová aplikace, neboť po vypnutí serveru by stejně neměly smysl. K čemu by nám například byla stará IP adresa a port zařízení, když po dalším připojení se může změnit?

ER digram znázorňuje strukturu databáze:




Obrázek 16 ERD

Základní tabulkou je tabulka users. Ta uchovává informace o uživatelích. Obsahuje id jako primární klíč, uživatelské jméno a hashované heslo uživatele. Dále pak tabulka conversations, která uchovává vytvořené konverzace, a tedy vytváří vztahy mezi uživateli, obsahuje id jako primární klíč a název. Každý uživatel může být členem mnoha konverzací a naopak konverzace může mít mnoho členů. Tuto M:N vazbu rozložíme pomocí další tabulky nazvané conversations_users, která uchovává id uživatele a id příslušné konverzace.

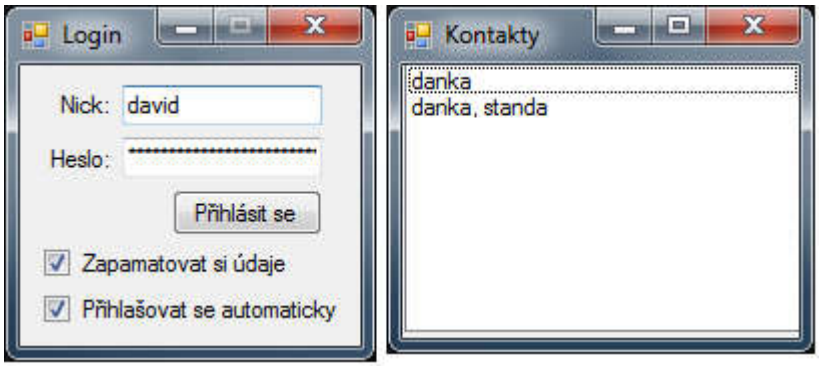
## 6.3 Požadavky na spuštění

Předně je požadován Microsoft Visual C++ Redistributable for Visual Studio 2015 (x86)[16]. Dále je potřeba Microsoft SQL Server 2014[12] a naimportovaná databáze „Chat“. Také je potřeba pro připojení k databázi v souboru Main.h změnit v připojovacím řetězci IP adresu a další parametry. Aplikace byla testována na Microsoft Windows 7 a Windows 10, běžet by měla na Windows Vista nebo vyšším.

[1] INVITATION datagram posílá knihovna pro podporu sítě, jsou velmi malé a slouží k nastavení NAT tabulky na routeru odesílatele a předpokládá se, že na routeru protistrany budou zahozeny.

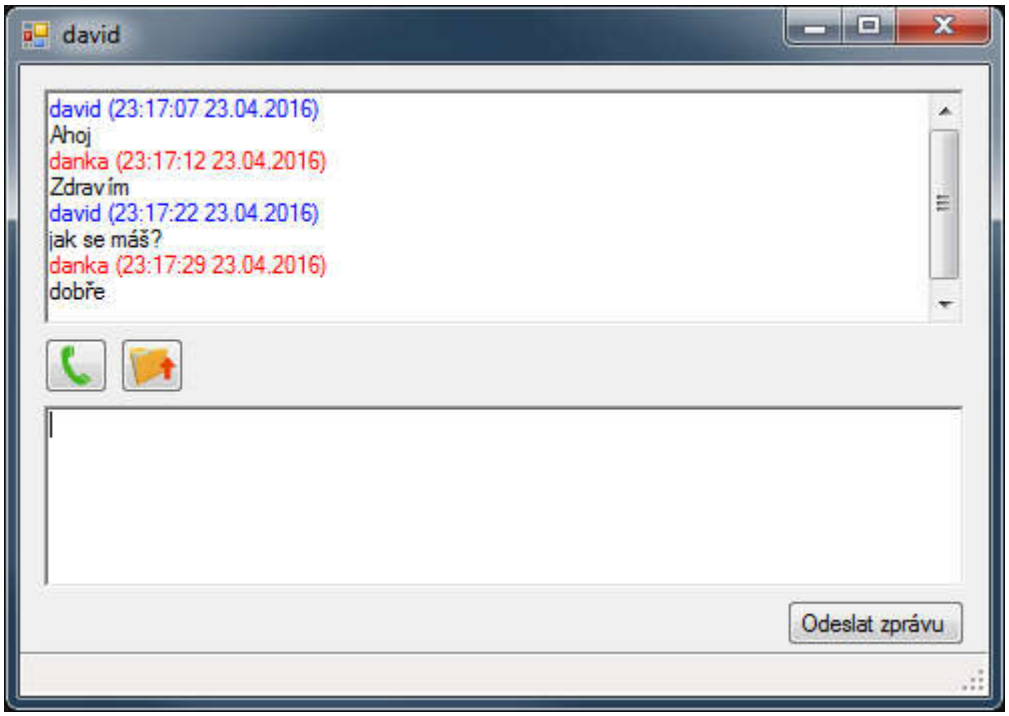
# 7 Klientská aplikace

Po spuštění aplikace se nejdříve zobrazí přihlašovací okno, do kterého uživatel vyplní uživatelské jméno a heslo. Poté stiskne tlačítko, které na základě vyhodnocení přihlašovacích údajů na serveru buď uživatele informuje o neexistujícím uživatelském jméně, chybně zadaném heslu, nebo se přihlašovací okno uzavře a objeví se okno se seznamem uživatelových kontaktů.




Obrázek 17 Klientovo okno přihlášení a kontaktní list

Dvojklikem na vybraný kontakt nebo skupinu se otevře chatovací okno, které zobrazuje přijaté zprávy a umožňuje posílat zprávy. Nachází se v něm tlačítka pro odesílání souborů, zahájení hovoru a zapnutí streamování videa.




Obrázek 18 Chatovací okno

Pro příjem a odesílání textových zpráv jsou vytvořeny dva kanály typu ConfirmatoryChannel. Tyto kanály existují po celou dobu trvání spojení.

## 7.1 Posílání souborů

Jakmile uživatel klikne na příslušné tlačítko v chatovacím okně, objeví se dialogové okno, ve kterém si má uživatel vybrat soubor, který chce odeslat. Poté dialogové okno zmizí a v chatovacím okně se zobrazí detailní informace o zasílaném souboru. Ty jsou také zaslány kanálem, který je, tak jako kanál pro chatování vytvořený po celou dobu trvání spojení, typu ConfirmatoryChannel.

Takto se tato žádost s názvem a velikostí souboru odešle protistraně, které se zobrazí okno, ve kterém se nacházejí dvě tlačítka. Jedno pro přijetí souboru a druhé pro jeho odmítnutí. V případě, že se uživatel rozhodne soubor přijmout, vytvoří se nový kanál typu MassChannel a vybraný soubor se začne přenášet. Odesílateli se zobrazí stejné okno. V těchto oknech se během přenosu zobrazuje průběh přenosu, tj. aktuální rychlost a počet přenesených dat.

## 7.2 Hovor

Hovor je vyvolán jedním z uživatelů konverzace stisknutím ikony v chatovacím okně. Poté se všem uživatelům v konverzaci objeví okno, ve kterém mohou ostatní uživatelé hovor přijmout nebo zamítnout.

Pro odebírání dat z mikrofonu, jejich enkódování a následné dekódování u druhého uživatele je využita knihovna FFmpeg[3]. Přehrávání zvuku pak zajišťuje knihovna libao[6]. Zvuk se enkóduje kodekem Speex[11], který má dobrý kompresní poměr pro přenos hlasu.

## 7.3 Video stream

V probíhajícím hovoru může uživatel kliknout na tlačítko, které spustí streamování videa z jeho kamery. Na rozdíl od streamování audia, které je možné provozovat i mezi více uživateli, streamování videa zatím funguje pouze mezi dvěma uživateli.

Pro odebírání dat z kamery, enkódování těchto dat a následné dekódování u protistrany je také použita knihovna FFmpeg[3]. Video se enkóduje kodekem H.264[18].

## 7.4 Požadavky na spuštění

V souboru Main.h ve funkci *connectToServerAndLogin* je potřeba nastavit IP adresu, na kterou se má klient připojit.

Ke spuštění je tak jako u serverové aplikace požadován Microsoft Visual C++ Redistributable for Visual Studio 2015 (x86)[16].

# 8 Testovací aplikace

V průběhu vývoje byly vytvořeny konzolové testovací aplikace sloužící k ladění různých služeb, které poskytuje klientská aplikace. Na rozdíl od cílových aplikací existuje vždy jen jedna aplikace, jelikož se nevyužívá STUN server. Spojení je tedy možné jen v LAN nebo i přes internet, ale minimálně na jedné straně musí mít uživatel otevřený port 9233 na svém routeru.

Proces spojení je u všech těchto aplikací stejný. Aplikace, která vyhledá ve stejném adresáři, ze kterého je spuštěna, soubor „destination.txt“, ve kterém se musí nacházet IP adresa např. „11.11.11.11“, a pokusí se navázat spojení s danou IP adresou. Komunikuje se přes UDP port 9233 v obou směrech.

Tyto aplikace ukazují, jak snadné je napojení knihovny, kterou samozřejmě využívají, na jakoukoli aplikaci.

**Aplikace na posílání souborů**

Po úspěšném spojení a po stisku klávesy ENTER odesílatelem aplikace vyhledá další soubor s názvem „file.txt“, ve kterém je napsána cesta k souboru, který chce uživatel odeslat. Cesta může být např. „C:\soubor.avi“. Poté se již spustí samotný přenos dat.

**Aplikace na streamování audia**

Jakmile dojde k úspěšnému spojení, může jeden z uživatelů stisknout klávesu ENTER. Tím se u něj spustí čtení dat z mikrofonu a streamování druhému uživateli. Druhý uživatel by měl slyšet streamovaný zvuk.

**Aplikace na streamování videa**

Postup před začátkem streamování je stejný jako u streamování audia. Rozdíl spočívá v tom, že druhému uživateli se otevře okno, ve kterém by měl vidět obraz z kamery prvního uživatele.

# Závěr

V této práci jsem se zaměřil na možnosti spojení dvou počítačů, kdy se oba počítače nacházejí za routerem s NATem a během procesu vytváření spojení mohou využít STUN serveru, tedy serveru, který jim pomůže spojení navázat. S využitím techniky UDP hole punching lze vždy dosáhnout úspěšného spojení.

V praktické části, ve které měla být vytvořena aplikace pro chatování, je popsána serverová aplikace, která funguje jako STUN server a také klientům poskytuje další služby, a klientská aplikace, která vytvořeného spojení mezi dvěma nebo i více uživateli využívá k poskytování služeb jako je zasílání textových zpráv, posílání souborů, VoIP a streamování videa z kamery.

Klientská i serverová aplikace využívá knihovnu, jenž byla také vytvořena jako součást této práce, a která zajišťuje nad protokolem UDP spolehlivost přenášených dat v závislosti na požadované službě. U posílání textových zpráv se stará o spolehlivé přenesení a správné pořadí. Při přenosu souborů je navíc kladen důraz na co největší rychlost samotného přenosu a minimalizaci režijních dat. U streamingu není důležité zajišťovat spolehlivost, ale stačí zajistit správné pořadí přenášených dat.

Všechny cíle této práce byly splněny a vzniklé aplikace, které bylo velmi obtížné naprogramovat i z toho důvodu, že byly programovány v C++, jsou funkční, velmi optimalizované a přesně odpovídají požadavkům na ně kladeným.

# Seznam použité literatury


[1] A Brief Description. *Cplusplus.com* [online]. [cit. 2016-04-07]. Dostupné z: http://www.cplusplus.com/info/description

[2] Cross Compare of SQL Server, MySQL, and PostgreSQL. *Postgres OnLine Journal* [online]. [cit. 2016-04-05]. Dostupné z: http://www.postgresonline.com/journal/archives/51-Cross-Compare-of-SQL-Server,-MySQL,-and-PostgreSQL.html

[3] *Ffmpeg* [online]. [cit. 2016-04-05]. Dostupné z: https://www.ffmpeg.org

[4] HAVRLANT, David. *Problémy s NAT a jejich řešení v multiplayer hrách*. Opava, 2014. Bakalářská práce. Slezská univerzita v Opavě. Vedoucí práce RNDr. Šárka Vavrečková, Ph.D.

[5] Introduction to SQL. *W3schools.com* [online]. [cit. 2016-04-05]. Dostupné z: http://www.w3schools.com/sql/sql_intro.asp

[6] *Libao* [online]. [cit. 2016-04-05]. Dostupné z: https://www.xiph.org/ao

[7] MEYERS, Scott. *Effective STL: 50 specific ways to improve your use of the standard template library*. Boston: Addison-Wesley, 2001. Addison-Wesley professional computing series. ISBN 02-017-4962-9.

[8] ROHÁČ, Michal. *Optimalizace protokolu TCP*. Ostrava, 2006. Dostupné z: www.cs.vsb.cz/grygarek/TPS/projekty/0506Z/roh035-TCP-Optimization.pdf. Semestrální práce. VŠB-TU v Ostravě.

[9] *Skype* [online]. [cit. 2016-04-05]. Dostupné z: https://www.skype.com/cs

[10] Slim Reader/Writer (SRW) Locks. *Msdn.microsoft.com* [online]. [cit. 2016-04-08]. Dostupné z: https://msdn.microsoft.com/en-us/library/windows/desktop/aa904937(v=vs.85).aspx

[11] *Speex* [online]. [cit. 2016-04-05]. Dostupné z: http://www.speex.org

[12] *SQL Server 2014* [online]. [cit. 2016-04-05]. Dostupné z: https://www.microsoft.com/cs-cz/server-cloud/products/sql-server

[13] *Telegram* [online]. [cit. 2016-04-05]. Dostupné z: https://telegram.org

[14] Transportní vrstva. *Gybon* [online]. [cit. 2016-04-05]. Dostupné z: http://www.gybon.cz/~rusek/vyuka/site/site007.html

[15] TYSON, Jeff. How Network Address Translation Works. *Howstuffworks* [online]. [cit. 2016-04-05]. Dostupné z: http://computer.howstuffworks.com/nat1.htm

[16] *Visual C++ Redistributable for Visual Studio 2015* [online]. [cit. 2016-04-05]. Dostupné z: https://www.microsoft.com/en-gb/download/details.aspx?id=48145

[17] *Visual Studio* [online]. [cit. 2016-04-05]. Dostupné z: https://www.visualstudio.com/

[18] What is H.264. *H264info.com* [online]. [cit. 2016-04-05]. Dostupné z: http://www.h264info.com/h264.html

[19] What is Network Address Translation? *Whatismyipaddress.com* [online]. [cit. 2016-04-05]. Dostupné z: http://whatismyipaddress.com/nat

[20] *Wireshark* [online]. [cit. 2016-04-05]. Dostupné z: https://www.wireshark.org

# Seznam tabulek

# Seznam obrázků

# Seznam zkratek

| | |
|---|---|
| NAT | Network Address Translation |
| STUN | Session Traversal Utilities for NAT |
| TCP | Transmission Control Protocol |
| UDP | User Datagram Protocol |

# Přílohy

Na přiloženém CD:

- text této práce (FPF_DP_16_Prima komunikace v pocitacove siti_Havrlant David.pdf)
- aplikace jako praktická část této práce (složka Aplikace)