VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

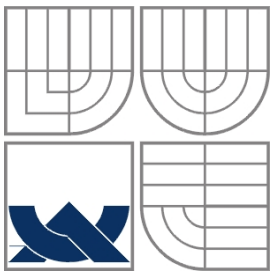
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

VIDEOKONFERENČNÍ SYSTÉM PRO VZDÁLENOU VÝUKU

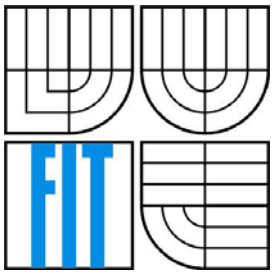
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE JOSEF HOLÁN
AUTHOR

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

VIDEOKONFERENČNÍ SYSTÉM PRO VZDÁLENOU VÝUKU

VIDEOCONFERENCIAL SYSTEM FOR REMOTE LEARNING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

JOSEF HOLÁN

VEDOUCÍ PRÁCE
SUPERVISOR

ING. PETR SCHWARZ

BRNO 2007

Abstrakt

Tato diplomová práce pojednává o možnostech streamování v Javě pomocí knihovny Java Media Framework. Pro vytvoření zadaného systému to znamená, na jedné straně vytvořit program, který bude snímat multimediální data z kamery a bude je vysílat do sítě. Na druhé straně bude program, který dokáže tyto data přijímat.

Předpokládá se jeden vyučující a skupina studentů. Obraz a zvuk přednášky je přenášen všem studentům. Komunikace studentů a vyučujícího se řeší textovým chatem a vyučující má možnost otevřít další audio nebo video kanál s jakýmkoliv studentem. Chat je řešen pomocí přenosu přes TCP/IP protokol. Zvuk (obraz) z tohoto kanálu může být dle volby přednášejícího přenášen i ostatním studentům.

Klíčová slova

Streaming, Java Media Framework, multicast, unicast, TCP/IP, UDP, videokonference, RTP.

Abstract

The main goal of this thesis is to design videoconferential system by Java Media Framework. This means to write two programs. One will capture multimedia data from webcam and will sent it into network. The second application which can receive this data is supposed to be on another computer. The multimedia data are transmitted by using Java Media Framework library.
We suppose one lector and group of students. Video and audio channel is transmitted to all students. Communication between lector and student is solved using chat. Lector has possibility open another audio or video channel with all students. Multimedia data will be transferred using Real-time Transport Protocol (RTP). The message including information about student is transferred by Transmission Control Protocol (TCP).

Keywords

Streaming, Java Media Framework, multicast, unicast, TCP/IP, UDP, videoconferential system, RTP.

Citace

Josef Holán: Videokonferenční systém pro vzdálenou výuku, diplomová práce, Brno, FIT VUT v Brně, 2007

Videokonferenční systém pro vzdálenou výuku

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Petra Schwarze, Ing. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Josef Holán
21.5.2007

Poděkování

Děkuji vedoucímu diplomové práce panu Ing. Petru Schwarzovi za cenné rady a vstřícný přístup při vypracování této práce.

Obsah

1 Úvod

Cílem diplomové práce bude seznámit čtenáře s možnostmi streamování v Javě a navrhnout systém pro vzdálenou výuku. To znamená jednak vytvořit program, který bude snímat multimediální data z kamery a vysílat je do sítě pomocí multicastu popsaného v kapitole 5. Na druhé straně bude druhá aplikace, která má být také součástí diplomové práce a jejímž účelem bude tato data přijímat. Přenosem mediálních dat se zabývá Java Media Framework (JMF). Tato knihovna je blíže popsaná v kapitole 3.

Předpokládá se jeden vyučující a skupina studentů. Obraz a zvuk přednášky je přenášen všem studentům. Komunikace studentů a vyučujícího se řeší textovým chatem a vyučující má možnost otevřít další audio/video kanál s jakýmkoliv studentem. Chat je řešen pomocí přenosu přes TCP/IP protokol. Zvuk (obraz) z tohoto kanálu může být dle volby přednášejícího přenášen i ostatním studentům.

Pro přenos multimediálních dat bude využit protokol RTP a s ním související protokol RTCP. Tyto dva protokoly jsou popsány v kapitole 4 a 5. Dotaz spolu s informacemi o studentovi budou posílány přes protokol TCP/IP, který je popsán v první kapitole.

Jak už bylo řečeno, systém by měl fungovat jako e-learning. Je to způsob předávání informací mezi lidmi prostřednictvím internetu. Různé učební texty mohou být zasílány například emailem. Avšak v dnešní době, díky zvyšujícím se rychlostem přenosu dat internetem, se stále rozšiřují nabídky videokonferencí a spolu s ní i různé způsoby přednášek konaných přes internet. Systém navržený v této práci by měl reprezentovat klasickou přednášku, kterou vede profesor ke svým studentům na vysoké škole. Hlavní výhodou tohoto systému je, že student může sledovat přednášku, aniž by se nacházel v místě konání přednášky. Zároveň i profesor může vést přednášku z jakéhokoliv místa, kde má možnost připojení k internetu s dostatečnou rychlostí.

2 TCP/IP

2.1 Architektura TCP/IP

Celosvětová síť Internet je v současnosti založena na protokolové sadě TCP/IP (Transmission Control Protocol). Její vývoj probíhal od počátku sedmdesátých let. Byla založena hlavně na zásadě vývoje TCP/IP od jednoduššího ke složitějšímu.

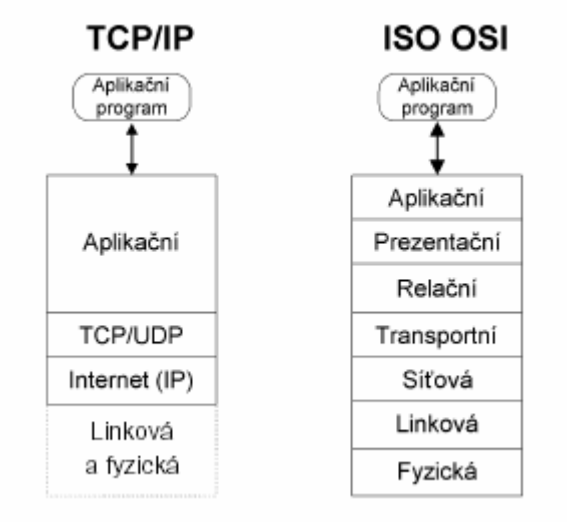
Síť nemusí být spolehlivá, musí však být co nejrychlejší. To znamená, že může docházet ke ztrátě paketů a spolehlivost si zajišťují až koncové uzly sítě, a to až na transportní či vyšší vrstvě, pokud je spolehlivost vyžadována. Pro zajištění spolehlivosti musí mít koncový uzel vyrovnávací paměti pro případ žádosti o opakování.

Upřednostňuje se nespojovaný charakter komunikace na úrovni sítě, tedy síť poskytuje nespojované a nespolehlivé služby. Spojovaný charakter komunikace si vytváří opět až koncový uzel sítě, je-li to nezbytné.

Vrstvový model TCP/IP neobsahuje vrstvy relační a presentační jako model OSI, neboť tyto služby těchto vrstev nejsou využívány všemi aplikacemi, a v takových případech zbytečně zvyšují režii přenosu a tedy užitečný přenosový výkon sítě. Sítě, které tyto služby vyžadují si je musí sami implementovat.

2.2 Vrstvová struktura TCP/IP

Problematika komunikace je z pohledu této sady rozdělena do čtyř vrstev (aplikační, transportní, síťová a vrstva síťového rozhraní) na rozdíl od systému OSI, který je sedmivrstvý jak můžeme vidět na obr.1.



Obr. 1: Porovnání síťových modelů TCP/IP a ISO OSI

Vrstva síťového rozhraní není blíže specifikována touto sadou, neboť je závislá na použité přenosové technologii (ETHERNET, TOKEN RING, ATM a další). Zajišťuje vysílání a příjem paketů ze sítě nebo do sítě.

Síťové vrstvě prakticky odpovídá IP protokol (Internet Protokol), proto je také tato vrstva často nazývána IP vrstva a zajišťuje směrování paketů po síti a poskytuje nespojovanou a nespolehlivou službu.

IP protokol přenáší tzv. IP datagramy mezi vzdálenými počítači. Každý IP datagram ve svém záhlaví nese adresu příjemce, což je úplná směrovací informace pro dopravu IP datagramu k adresátovi. Takže síť může přenášet každý IP datagram samostatně. IP datagramy tak mohou k adresátovi dorazit i v jiném pořadí než byly odeslány. Každé síťové rozhraní v rozsáhlé síti Internet má svou celosvětově jednoznačnou IP adresu (jedno síťové rozhraní může mít více IP adres, avšak jednu IP adresu nesmí používat více síťových rozhraní). IP protokol v sobě zahrnuje další protokoly pomocí nichž jsou realizovány funkce této vrstvy. Jsou to protokoly: ICMP (Internet Control Message Protocol), ARP (Address Resolution Protocol) a RARP (Reverse ARP), OSPF (Open Shortest Path First) a IGMP (Internet Group Management Protocol).

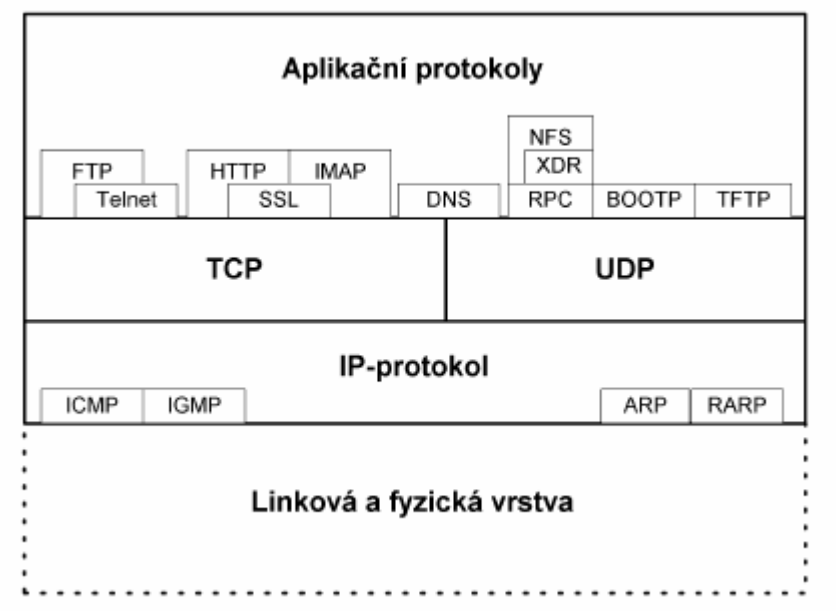
Transportní vrstva realizuje a zajišťuje komunikaci koncových uzlů. Multiplexuje(ve směru do sítě) a demultiplexuje (ve směru od sítě) datový tok od jednotlivých aplikací k jiným aplikacím. Tato vrstva je prakticky tvořena protokoly UDP a TCP.

Protokol TCP dopravuje data pomocí TCP segmentů, které jsou adresovány jednotlivým aplikacím. Protokol UDP dopravuje data pomocí tzv. UDP datagramů. Protokoly TCP a UDP zajišťují spojení mezi aplikacemi běžícími na vzdálených počítačích.a mohou zajišťovat i komunikaci mezi procesy běžícími na témže počítači (podrobněji budou popsány v kapitolách 2.3 a 2.4). Rozdíl mezi protokoly TCP a UDP spočívá v tom, že protokol TCP je tzv. spojovanou službou, tj. příjemce

potvrzuje přijímaná data. V případě ztráty dat (ztráty TCP segmentu) si příjemce vyžádá zopakování přenosu. Protokol UDP přenáší data pomocí datagramů (obdoba telegramu), tj. odesílatel odešle datagram a už se nezajímá o to, zdali byl doručen. Adresou je tzv. port.

Aplikační vrstva TCP/IP je redukcí vrstev relační, prezentační a aplikační vrstvy modelu OSI jak můžeme vidět na obr. 1. Aplikačních protokolů je velké množství. Z praktického hlediska je lze rozdělit na uživatelské a služební.

Uživatelské protokoly jsou ty, které využívají uživatelské aplikace (např. pro vyhledávání informací v Internetu). Příkladem takových protokolů jsou protokoly: HTTP (HyperText Transfer Protokol), SMTP (Simple Mail Transfer Protocol), Telnet (vzdálený přístup), FTP (File Transfer Protocol), IMAP(Internet Message Access Protocol), POP3 (Post Office Protocol version 3), DHCP (Dynamic Host Configuration Protocol) atd. Služební protokoly jsou ty protokoly, se kterými se běžní uživatelé Internetu nesetkají. Tyto protokoly slouží pro správnou funkci Internetu. Jedná se např. o směrovací protokoly, které používají směrovače mezi sebou, aby si správně nastavily směrovací tabulky. Dalším příkladem je protokol SNMP (Simple Network Management Protocol), který slouží ke správě sítí.



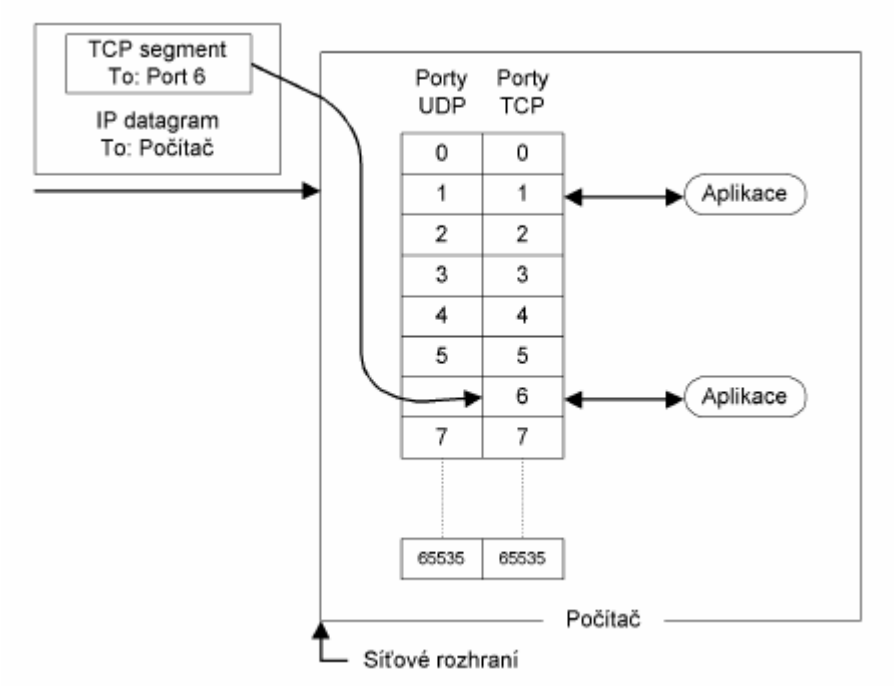
Obr. 2: Některé protokoly sady TCP/IP a jejich pozice v modelu TCP/IP

2.3 TCP

Protokol TCP je proti protokolu IP protokolem vyšší vrstvy. Zatímco protokol IP přepravuje data mezi libovolnými počítači v Internetu, tak protokol TCP dopravuje data mezi dvěma konkrétními aplikacemi běžícími na těchto počítačích. Pro dopravu dat mezi počítači se využívá protokol IP. Protokol IP adresuje IP-adresou pouze síťové rozhraní počítače (často se používá přirovnání k běžnému poštovnímu styku, pak IP-adresa odpovídá adrese domu a port (adresa v protokolu TCP) odpovídá jménu konkrétního obyvatele domu).

Protokol TCP je spojovanou službou (*connection oriented*), tj. službou která mezi dvěma aplikacemi naváže spojení (vytvoří na dobu spojení virtuální okruh). Tento okruh je plně duplexní (data se přenášejí současně na sobě nezávisle oběma směry). Přenášené bajty jsou číslovány. Ztracená nebo poškozená data jsou znovu vyžádána. Přenos všech dat je zabezpečen kontrolním součtem, tedy aplikace používající protokol TCP si nemusí dělat starosti s tím, zdali náhodou nebyla nějaká data během přenosu ztracena nebo díky chybě přenosu upravena. Toto zabezpečení je účinné pouze proti poruchám technických prostředků. Nezabezpečuje tedy data proti útočníkům, kteří mohou data pozměnit a současně také přepočítat kontrolní součet. Ochranou přenášených dat proti takovýmto cíleným útokům se v rodině protokolů TCP/IP zabývají např. protokoly SSL, S/MIME.

Konce spojení (“odesílatel” a „adresát”) jsou určeny tzv. číslem portu. Toto číslo je dvojbajtové, takže může nabývat hodnot 0 až 65535. U čísel portů se často vyjadřuje okolnost, že se jedná o porty protokolu TCP tím, že se za číslo napíše lomítko a název protokolu (tcp). Pro protokol UDP je jiná sada portů než pro protokol TCP (též 0 až 65535), tj. např. port 53/tcp nemá nic společného s portem 53/udp (viz. obr. 3). Cílová aplikace je v Internetu adresována (jednoznačně určena) IP-adresou, číslem portu a použitým protokolem (TCP nebo UDP). Protokol IP dopraví IP-datagram na konkrétní počítač. Na tomto počítači běží jednotlivé aplikace. Podle čísla cílového portu operační systém pozná které aplikaci má TCP segment doručit.

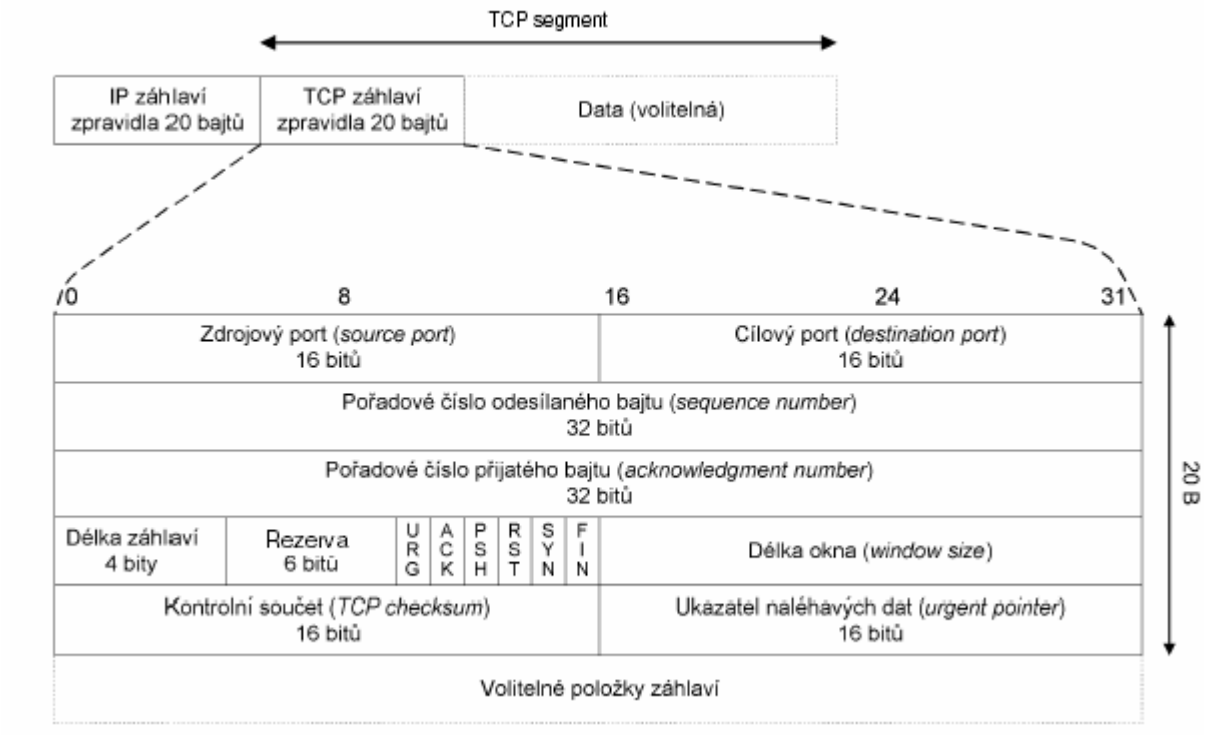


Obr. 3: Porty TCP a UDP

Základní jednotkou přenosu v protokolu TCP je TCP paket (segment). Aplikace běžící na jednom počítači posílá protokolem TCP data aplikaci běžící na jiném počítači. Aplikace potřebuje přenést např. soubor velký 2 GB. Jelikož TCP segmenty jsou baleny do IP datagramů, který má pole dlouhé 16 bitů, tak TCP segment může být dlouhý maximálně 65535 minus délka TCP-záhlaví. Přenášené 2 GB dat musí být rozděleny na segmenty, které se vkládají do TCP paketu – přeneseně se pak místo TCP paket říká TCP segment. TCP segment se vkládá do IP-datagramu. IP-datagram se vkládá do linkového rámce. Použije-li se příliš velký TCP-segment, který se celý vloží do velkého IP-datagramu, který je větší než maximální velikost přenášeného linkového rámce (MTU), pak IP protokol musí provést fragmentaci IP-datagramu. Cílem je vytvářet segmenty takové velikosti, aby fragmentace nebyla nutná.

2.3.1 TCP segment

TCP segment má strukturu uvedenou na obr. 4.



Obr. 4: TCP Segment

Zdrojový port (*source port*) je port odesílatele TCP segmentu, cílový port (*destination port*) je portem adresáta TCP segmentu. Zdrojový port, cílový port, zdrojová IP-adresa, cílová IP-adresa a protokol (TCP) jednoznačně identifikuje v daném okamžiku spojení v Internetu.

TCP segment je část z toku dat přenášených od odesílatele k příjemci. Pořadové číslo odesílaného bajtu je pořadové číslo prvního bajtu TCP segmentu v toku dat od odesílatele k příjemci (TCP segment nese bajty od pořadového čísla odesílaného bajtu až do délky segmentu). Tok dat v opačném směru má samostatné (jiné) číslování svých dat. Jelikož pořadové číslo odesílaného bajtu je 32 bitů dlouhé, tak po dosažení hodnoty 2^{32}-1 nabude cyklicky opět hodnoty 0. Číslování obecně nezačíná od nuly (ani od nějaké určené konstanty), ale číslování by mělo začínat od náhodně zvoleného čísla. Vždy když je nastaven příznak SYN, tak operační systém odesílatele začíná znovu číslovat, tj. vygeneruje startovací pořadové číslo odesílaného bajtu, tzv. ISN (*Initial Sequence Number)*.

Naopak pořadové číslo přijatého bajtu vyjadřuje číslo následujícího bajtu, který je příjemce připraven přijmout, tj. příjemce potvrzuje, že správně přijal vše až do pořadového čísla přijatého bajtu mínus jedna.

Délka záhlaví vyjadřuje délku záhlaví TCP segmentu v násobcích 32 bitů (4 bajtů) – podobně jakou IP-záhlaví.

Délka okna vyjadřuje přírůstek pořadového čísla přijatého bajtu, který bude příjemcem ještě akceptován

Ukazatel naléhavých dat může být nastaven pouze v případě, že je nastaven příznak URG. Přičte-li se tento ukazatel k pořadovému číslu odesílaného bajtu, pak ukazuje na konec úseku naléhavých dat. Odesílatel si přeje, aby příjemce tato naléhavá data přednostně zpracoval. Tento mechanismus používá např. protokol Telnet.

Aplikační protokol Telnet přenáší data mezi terminálem a serverem. Kromě běžných aplikačních dat, může tok dat obsahovat i příkaz IAC (následující data interpretuje jako příkaz protokolu Telnet). Příkaz IAC začíná znakem IAC (desítkově 255) následovaný příslušným příkazem. Jako příkaz může být např. ABORT (zruš proces), který je reprezentován bajtem o desítkové hodnotě 238.

Příkazem ABORT signalizuje uživatel žádost o zrušení procesu. Uživatel například odeslal na server velké množství dat, která čekají ve vyrovnávací paměti serveru na zpracování. Pokud uživatel chce proces ukončit bez čekání na ukončení zpracování dat a nezpracovaná data zahodit, pak vyšle příkaz ABORT. Pokud by server zpracovával veškerá data sekvenčně, pak by se příkaz ABORT vykonal až po zpracování všech dat. Uživatel však chce proces ukončit okamžitě. V TCP segmentu nesoucím příkaz ABORT se nastaví příznak URG a vyplní se ukazatel naléhavých dat ukazující na příkaz ABORT.

Server podle příznaku URG zjistí, že TCP segment obsahuje naléhavá data, přičtením ukazatele naléhavých dat k pořadovému číslu odeslaného bajtu zjistí konec naléhavých dat. Nyní začne procházet vyrovnávací paměť od konce naléhavých dat směrem k počátku vyrovnávací paměti až narazí na bajt obsahující IAC. Nyní již má zjištěna celá naléhavá data a může je začít interpretovat, tj. v našem případě zrušit proces.

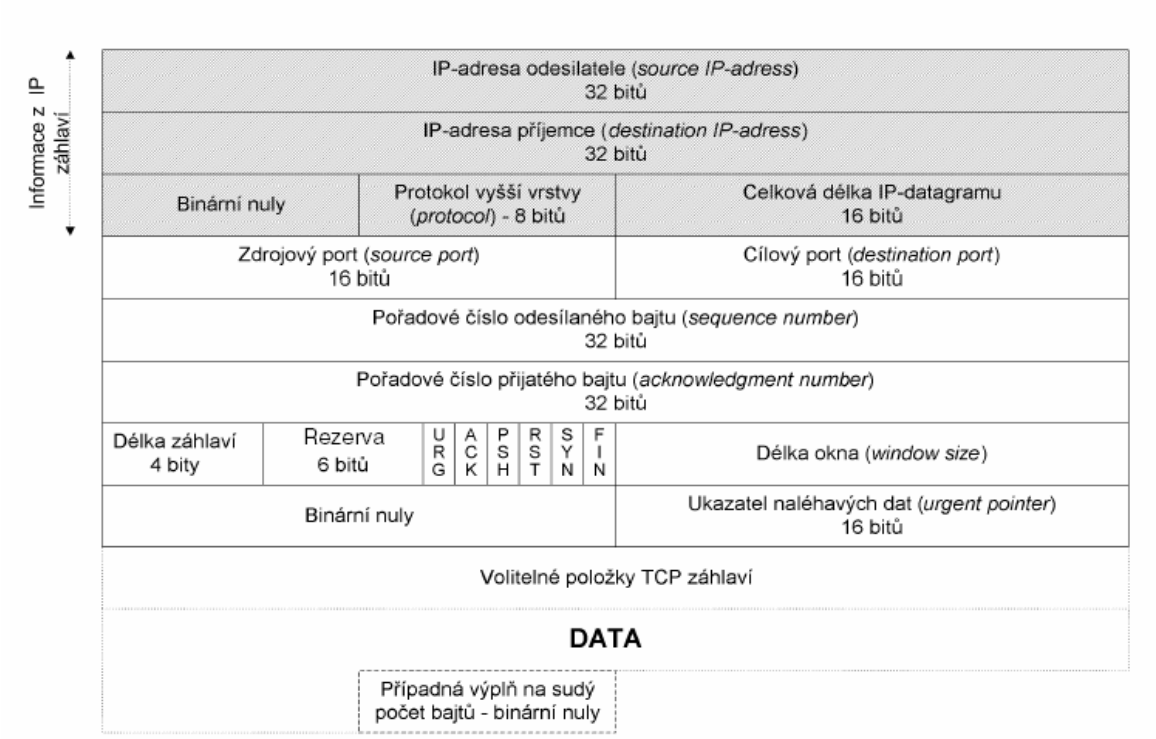
Využití ukazatele naléhavých dat závisí na tvůrci aplikace. Tvůrce většinou vyvíjí jak software pro odesílatele, tak i software pro příjemce. Většina aplikací tento mechanismus nevyužívá. Existují však i aplikace, jež ukazatel naléhavých dat využívají, ale ten ukazuje na počátek naléhavých dat (nikoliv na konec, jak určuje norma). V poli příznaků mohou být nastaveny následující příznaky:

- URG – TCP segment nese naléhavá data.
- ACK – TCP segment má platné pole „Pořadové číslo přijatého bajtu" (nastaven ve všech segmentech kromě prvního segmentu, kterým klient navazuje spojení).
- PSH – Zpravidla se používá k signalizaci, že TCP segment nese aplikační data, příjemce má tato data předávat aplikaci. Použití tohoto příznaku není ustáleno.
- RST – Odmítnutí TCP spojení.

- SYN – Odesílatel začíná s novou sekvencí číslování, tj. TCP segment nese pořadové číslo prvního odesílaného bajtu (ISN).
- FIN – Odesílatel ukončil odesílání dat. Pokud bychom použili přirovnání k práci se souborem, pak příznak FIN odpovídá konci souboru (EOF). Přijetí TCP segmentu s příznakem FIN neznamená, že v opačném směru není dále možný přenos dat. Jelikož protokol TCP vytváří plně duplexní spojení, tak příznak FIN způsobí jen uzavření přenosu dat v jednom směru. V tomto směru už dále nebudou odesílány TCP segmenty obsahující příznak PSH (nepočítaje v to případné opakování přenosu).

Kontrolní součet IP-záhlaví se počítá pouze ze samotného IP-záhlaví. Z hlediska zabezpečení integrity přenášených dat je důležitý kontrolní součet v záhlaví TCP-segmentu počítaný i z přenášených dat. Tento kontrolní součet se počítá nejen ze samotného TCP segmentu, ale i z některých položek IP-záhlaví. Kontrolní součet vyžaduje sudý počet bajtů, proto v případě lichého počtu se data fiktivně doplní jedním bajtem na konci.

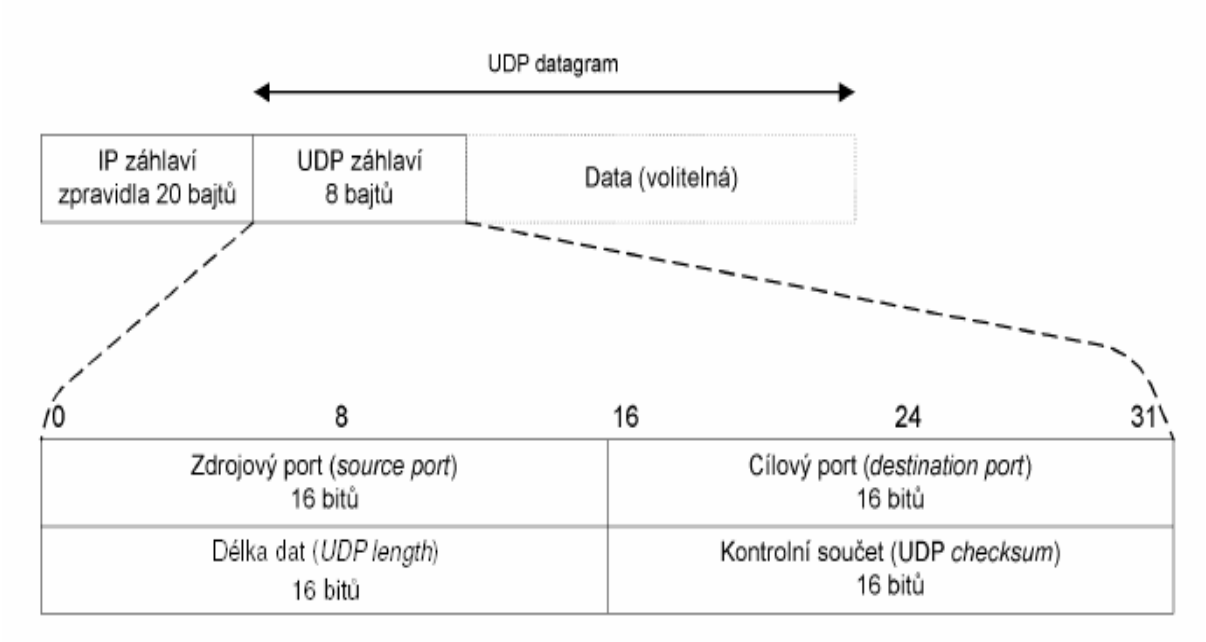
Kontrolní součet se počítá z polí znázorněných na obr. 5. Tato struktura slouží pouze pro výpočet kontrolního součtu (žádná takováto struktura nepřenáší).



Obr. 5: Pole, ze kterých se počítá kontrolní součet TCP záhlaví

2.4 UDP

Protokol UDP je jednoduchou alternativou protokolu TCP. Protokol UDP je nespojovaná služba (na rozdíl od protokolu TCP), tj. nenavazuje spojení. Odesílatel odešle UDP datagram příjemci a už se nestará o to, zdali se datagram náhodou neztratil (o to se musí postarat aplikační protokol). UDP datagramy jsou baleny do IP-datagramu, jak je znázorněno na obr. 6.



Obr. 6: Záhlaví UDP datagramu

Z předchozího obrázku je patrné, že záhlaví UDP protokolu je velice jednoduché. Obsahuje čísla zdrojového a cílového portu (To je úplně stejné jako u protokolu TCP). Jak už bylo jednou zmíněno, čísla portů protokolu UDP nesouvisí s čísly portů protokolu TCP. Protokol UDP má svou nezávislou sadu čísel portů. Pole délka dat obsahuje délku UDP datagramu (délku záhlaví + délku dat). Minimální délka je tedy 8, tj. UDP datagram obsahující pouze záhlaví a žádná data.

Zajímavé je že pole kontrolní součet nemusí být povinně vyplněné. Výpočet kontrolního součtu je tak v protokolu UDP nepovinný. V minulosti bylo u některých počítačů zvykem výpočet kontrolního součtu vypínat. Důvodem bylo zrychlení odezvy počítače. Zejména u důležitých serverů je třeba vždy zkontrolovat, zdali je opravdu výpočet kontrolního součtu zapnut. Nejnebezpečnější je to v případě DNS serveru, protože kontrolní součet pak je počítán jen na linkové vrstvě, ale např. linkový protokol SLIP výpočet kontrolního součtu také nepočítá, takže i technická porucha může způsobit poškození aplikačních dat, aniž by měl příjemce šanci to zjistit.

3 Java Media Framework (JMF)

V programovacím prostředí Javy existuje knihovna Java Media Framework (JMF), která umožňuje snadné zpracování časově závislých médií, jakými jsou např. obrazová či zvuková data vykazující určitou změnu v čase. Například u pohyblivých obrazů se časová závislost projevuje změnou obrazové scény.

JMF se svými metodami pokrývá širokou oblast multimediálního zpracování: získávání (akvizice) médií, přes jejich kompresi, přenos, aplikaci speciálních efektových algoritmů až k jejich prezentaci či archivaci. Využívá k tomu specializované nástroje, jako jsou multiplexory (slučovače) a demultiplexory (rozdělovače), kodéry a dekodéry, efektové procesory a renderery (prezentační nástroje).

K uskutečnění živého vysílání (vytvoření videokonference pres Internet nebo Intranet) je třeba mít možnost příjmu a vysílání multimediálních dat v reálném čase. Jejich zdrojem je například internetové rádio nebo webová kamera.

Přenos takových dat v reálném čase vyžaduje výkonnou komunikační sít. To znamená, ze musí splňovat vysoké nároky na přenosovou rychlost, zpoždění a kvalitu služeb. To je právě hlavní rozdíl mezi přenosem statických dat a přenosem dat v reálném čase. Proto protokoly užívané běžně pro přenos statických dat nevyhovují přenosům dat v reálném čase.

JMF pro přenos dat používá protokol RTP. Protokol RTP slouží k distribuci a příjmu multimediálních dat v reálném čase a je nezávislý na typu přenosové sítě a protokolu. Z důvodu povahy dat je protokol provozován nad UDP, což je nespolehlivý protokol.

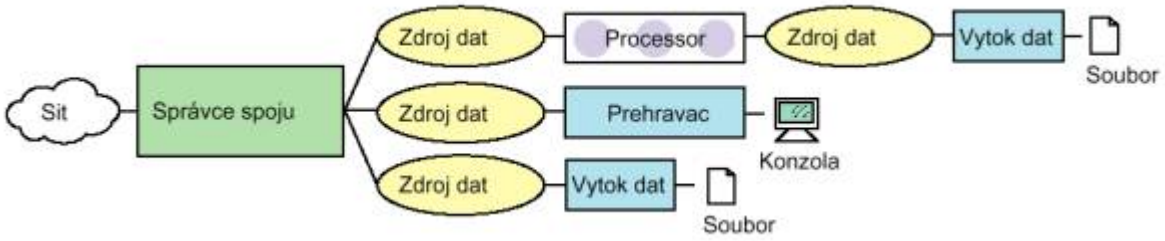
Protokoly, jenž jsou vhodné pro spolehlivý přenos dat v síti s malou šířkou pásma, jsou protokoly HTTP a FTP. Jsou založeny na protokolu TCP/IP. Zde platí, že pokud je paket ztracen či poškozen, dojde k jeho opětnému vyslání.

Z tohoto důvodu se pro přenos v reálném čase používají jiné protokoly než TCP. Jeden z nejčastěji používaných je protokol UDP (User Datagram Protocol). UDP je nespolehlivý protokol, který nezaručuje, aby každý vyslaný paket byl doručen k cíli. Dokonce nezaručuje ani to, aby pakety dorazily ve správném poradí.

JMF podporuje čtení i zápis nejrozšířenějších a nejčastěji používaných formátů souborů, jako jsou například čistě zvukové formáty AIFF, AU, MIDI, MP3 a WAV, nebo s pohyblivým obrazem kombinované formáty MPEG, QuickTime a AVI. Pro plynulý přenos médií v komunikační síti nabízí tato knihovna implementaci protokolu RTP/RTCP s možností nastavení jednotlivých parametrů přenosu. Umožňuje přitom kódování pohyblivých obrazových dat standardy MPEG i H.261.

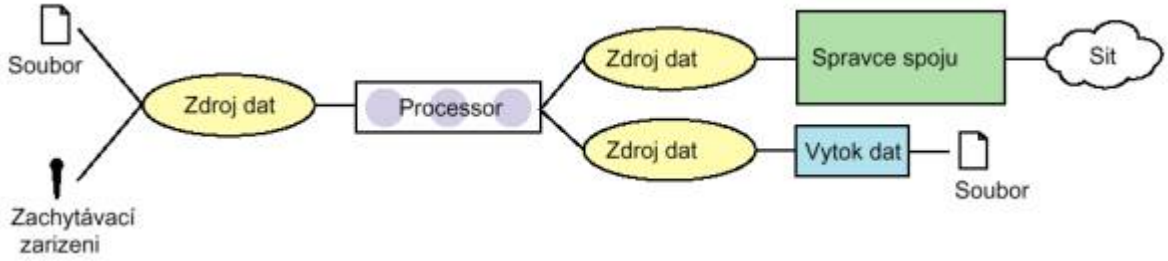
JMF umožňuje zachytávání a vysílání RTP datových toků přes API definované v balících javax.media.rtp, javax.media.rtp.event a javax.media.rtp.rtcp.

Příchozí RTP datové toky je možno lokálně přehrávat, ukládat je do souborů nebo obojí.



Obr. 7: Příjem RTP

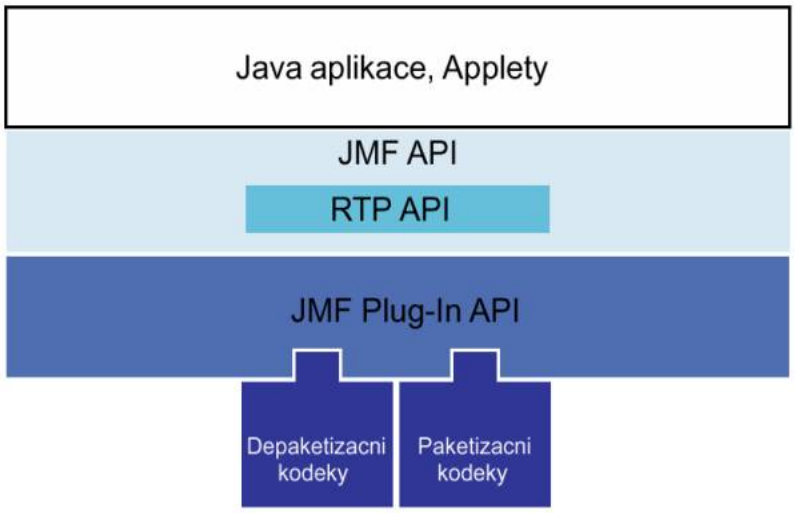
Odcházející toky mohou být rovněž lokálně přehrávány a ukládány do souborů nebo obojí.



Obr. 8: Vysílání RTP

3.1 JMF RTP architektura

JMF RTP API jsou navrženy tak, aby pracovaly nepřerušovaně se zachytávajícími, prezentačními a zpracovávacími částmi JMF. Přehrávače a procesory jsou užity pro prezentaci a manipulaci RTP mediálního toku stejně jako s jiným médiem. Je možno vysílat multimediální datové toky, jenž byly získány z lokálního zachytávajícího zařízení nebo ty, jenž byly získány z uloženého souboru.



Obr. 9: JMF RTP architektura

3.2 Managers

JMF API se skládá z rozhraní, která definují chování a interakce objektů užitých k zachytávání, zpracování a zobrazování časově závislých médií. Implementace těchto rozhraní je součást JMF. Tyto zprostředkující objekty se nazývají **managers**. Je jednoduché integrovat do JMF nové implementace klíčových rozhraní, které mohou být používány na místo existujících tříd. JMF používá tyto čtyři managery:

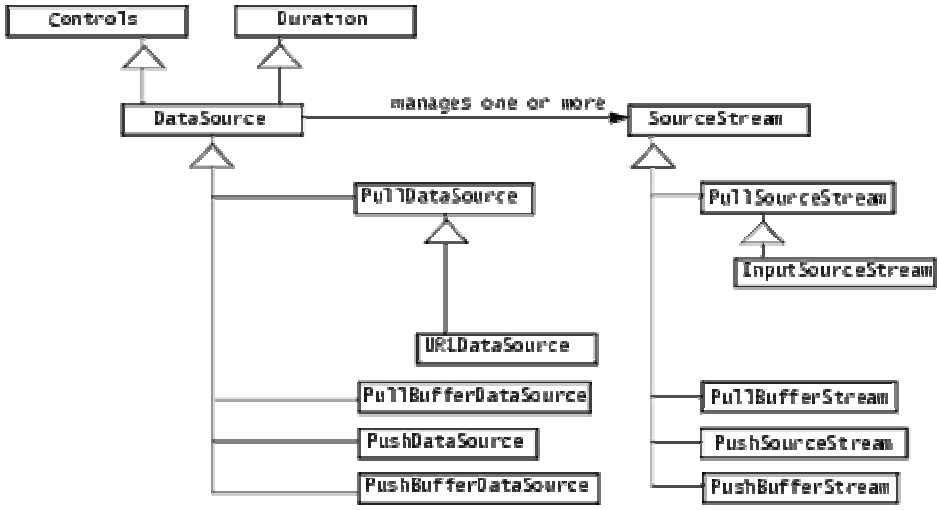
- **Manager** – ovládá objekty *Player*, *Processor*, *DataSource* a *DataSink*. Tato úroveň nepřímo umožňuje nové implementace, které mohou být jednoduše integrovány s JMF.
- **PackageManager** – obsahuje seznam balíčků, které obsahují JMF třídy jako jsou *Player*, *Processor*, *DataSource* a *DataSink*.
- **CaptureDeviceManager** – obsahuje seznam použitelných zachytávacích zařízení.
- **PlugInManager** – obsahuje seznam dostupných JMF plug-in procesních komponent, jako jsou *Multiplexory*, *Demultiplexory*, *Kodeky*, *Efekty* a *Renderery*.

Pro psaní programů, založených na JMF, potřebujeme použít dané *create* metody pro vytvoření instanci *Playeru*, *Processoru*, *DataSourcu* nebo *DataSinku* pro danou aplikaci. Pokud chceme zachytávat multimediální data z nějakého vstupního zařízení musíme použít *CaptureDeviceManager* pro nalezení použitelných zařízení a přístupových informací o nich. Chceme-li rozšířit funkcionalitu JMF implementováním nových plug-inů, musíme je zaregistrovat pomocí *PlugInManageru* pro zpřístupnění procesorům. Pro použití upraveného *Playeru*, *Procesoru*, *DataSourcu* nebo *DataSinku* s JMF, musíme zaregistrovat náš unikátní prefix daného upraveného objektu pomocí *PackageManageru*.

3.3 Data Model

JMF přehrávače médií obvykle používají *DataSource* k řízení přenosu obsahu dat. *DataSource* zapouzdřují umístění média a protokolu. Jakmile je *DataSource* používán, nemůže být použit ničím jiným.

DataSource je určen výhradně *MediaLocatorem* nebo *URL* (Universal Resource Locator). *MediaLocator* je podobný *URL* a může být z *URL* vytvořen. *DataSource* spravuje *SourceStream* objekty. Standardní datový zdroj používá bajtové pole jako přenosovou jednotku. *B*ufferDataSource používá buffer objekty jako jeho přenosovou jednotku. JMF definuje několik typů *DataSource* objektů:



Obr. 10: JMF data model

3.4 Push a Pull Data Sources

Média mohou být získána z různých zdrojů, jako jsou lokální nebo síťové soubory a živá vysílání. JMF zdroje dat mohou být rozděleny podle toho, jak je přenos zahájen:

- **Pull Data-Source** - klient iniciuje datový přenos a ovládá tok dat z *Pull Data-Source.* Existující protokoly pro tento typ dat jsou HTTP (Hypertext Transfer Protocol) a FILE. JMF definuje dva typy pull data sourců: *PullDataSource* a *PullBufferDataSource*, který používá *Buffer* objekty jako vlastní přenosovou jednotku.
- **Push Data-Source** - server iniciuje datový přenos a ovládá datový tok z *Push Data-Source. Push Data-Sources* obsahují broadcast média, multicast média, a VOD (video-on-demand). Pro broadcast data je jeden protokol RTP (Real-time transport protocol). *MediaBase* protokol je jeden protokol používaný pro VOD. JMF definuje dva typy *Push Data-Source: PushDataSource* a *PushBufferDataSource*, které používají *Buffer* objekty jako jednotky přenosu.

Stupeň ovládání, který může program poskytnout uživateli, záleží na typu zdrojových dat, která jsou prezentována. Např. MPEG soubor může být přemístěn a klientský program dovolí uživateli znovu přehrát videoklip nebo vyhledávat novou pozici a přehrát. Naproti tomu broadcast média jsou ovládány serverem a nemohou být přemístěna. Některé VOD protokoly mohou podporovat omezenou kontrolu uživatelem – např. klientský program může umožnit uživateli vyhledat novou pozici, ale neumožní rychlé přehrátí vpřed či vzad.

3.5 Speciální DataSourcy

JMF definuje dva typy speciálních datových zdrojů, klonovatelné *DataSource* a spojovací *DataSource*.

Klonovatelné *DataSource* implementuje *SourceCloneable* rozhraní, které definuje právě jednu metodu, *createClone()*. Zavoláním metody *createClone()* můžeme vytvořit libovolný počet klonů *DataSource*, které byly použity pro vytvoření klonovatelného *DataSource*.

Klony nemusí mít nutně stejné vlastnosti jako klonovatelné *DataSource* použité k jejich vytvoření z originálního *DataSource*. Např. klonovatelný DataSource vytvořený pro zachytávací zařízení, může fungovat jako master DataSource pro jeho klony.

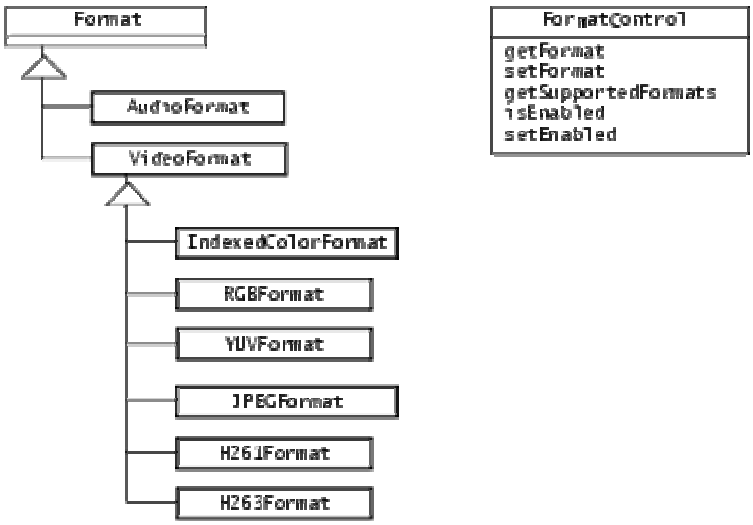
K vytvoření spojovacího *DataSource*, se zavolá v *Manageru* metoda *createMergingDataSource* a předá pole obsahující *DataSourcy*, které chceme spojit. Pro spojení je

nutné, aby všechny DataSourcy byly stejného typu; např., nemůžeme spojit *PullDataSource* a *PushDataSource*. Doba spojení DataSource je maximum z doby trvání *DataSource* objektů.

3.6 Formáty dat

Formát média je v JMF reprezentován objektem *Format*. Formát v sobě nenese žádné kódovací parametry nebo časové informace. Popisuje pouze název kódovacího formátu a data které daný formát potřebuje pro práci s médiem.

JMF rozšiřuje *Format* pro definici audio a video formátů.



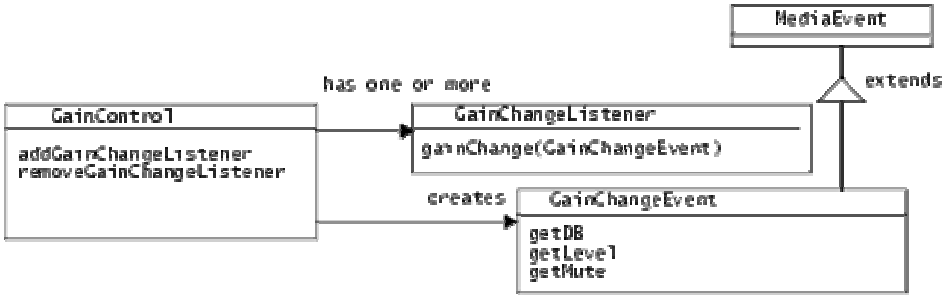
Obr. 11: JMF media formáty.

AudioFormat popisuje atributy specifické pro audio formát jako je vzorkovací frekvence, bitů za vzorek a počet kanálů. *VideoFormat* zapouzdřuje informaci relevantní k video formátu. Několik formátů je odvozeno z třídy *VideoFormat* pro popis atributů pro dané formáty. Jsou to tyto:

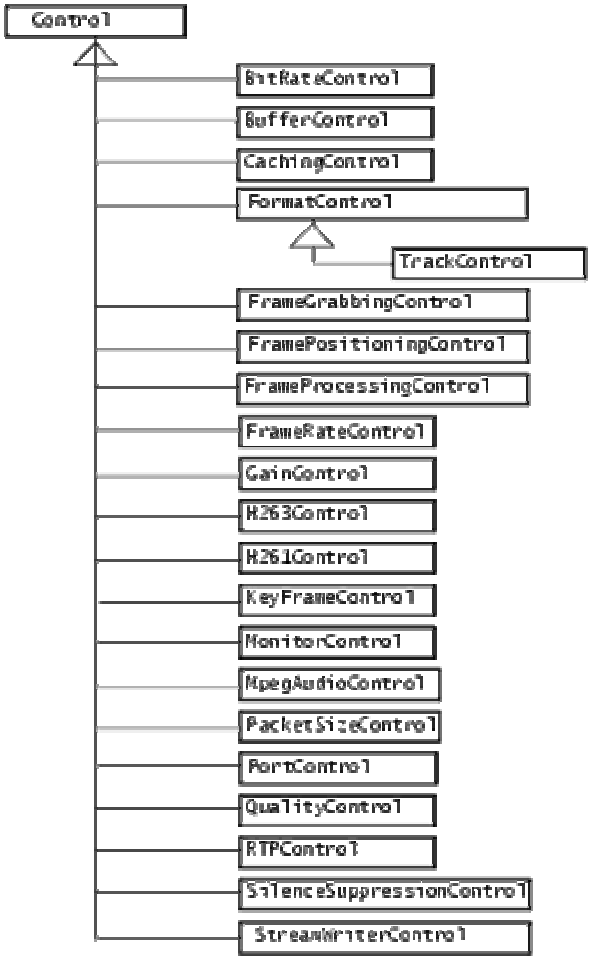
- IndexedColorFormat
- RGBFormat
- YUVFormat
- JPEGFormat
- H261Format
- H263Format

3.7 Standardní kontrolery

JMF definuje standardní třídu *Control* pro defaultní definici kontrolerů. *CachingControl* povoluje zobrazení a monitorování průběhu stahování daného média. *GainControl* povoluje zobrazení ovládání hlasitosti, jako je hlasitost nebo zapínání a vypínaní výstupu.



Obr. 12: Gain control



Obr. 13: JMF kontrolery.

Datasink nebo *Multiplexor* objekty, které čtou media z *DataSource* a zapisují je do nějakého výstupu jako je soubor, mohou implementovat *StreamWriterControl* rozhraní. Tento kontroler umožňuje uživateli limitovat velikost vytvořeného streamu.

FramePositioningControl a *FrameGrabbingControl* objekty umožňují objektům *Player* a *Processor* pracovat se snímky daného média, zejména videa.

Objekty, které mají parametr typu *Format* mohou používat *FormatControl* rozhraní k přístupu k tomuto parametru. *FormatControl* také umožňuje nastavení formátu pro dané médium.

JMF také definuje další kontrolery kodeků pro kontrolu hardwarových a softwarových enkodérů a dekodérů:

- **BitRateControl**
- **FrameProcessingControlFrameRateControl**
- **H261Control**
- **H263Control**
- **MpegAudioControl**
- **QualityControl**
- **SilenceSuppressionControl**

3.8 Komponenty uživatelského rozhraní

Ovládání UI může poskytnout přístup k uživatelskému rozhraní *Componet,* který poskytuje kontrolu nad chováním koncovému uživateli. K získání součásti standardního uživatelského rozhraní pro konkrétní *Control*, se zavolá metoda *getControlComponent*(). Tato metoda vrací AWT komponentu, jež umožňuje přidání vašeho místa zobrazení appletu nebo okna aplikace.

Kontroler může také poskytnout přístup k uživatelskému rozhraní *Components*. Například, Player poskytuje přístup k vizuální komponentě a ovládacímu panelu komponent – pro získání těchto součástí, zavoláme metody *getVisualComponent()* a *getControlPanelComponent()*.

Jestliže nechceme použít standardní ovládací komponenty, můžeme naimplementovat naše vlastní a použít listener k jejich ovládání. Například, můžeme implementovat naše vlastní GUI komponenty, které podporují interakci uživatele a *Playeru*. Akce na naše GUI komponenty spouští volání příslušných metod *Playeru*, jako *start* a *stop*.

3.9 Rozšířitelnost

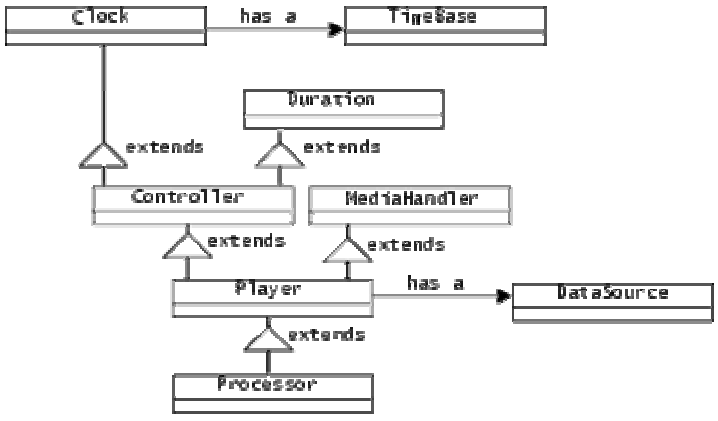
JMF funkcionalitu můžeme rozšířit dvěmi možnostmi:

- Implementováním vlastních procesních komponent (pluginů), které dokáží spolupracovat se standardním JMF Procesorem.
- Přímým naimplementováním tříd *Controller, Player, Processor, DataSource,* nebo *DataSink*.

Toho se využívá, pokud chceme přímo přistupovat a pracovat se zdroji dat, které mají formát se kterými původní JMF neumí pracovat jako jsou např. Microsoft's Media Player, Real Network's RealPlayer atd.

3.10 Zobrazení

JMF API definuje dva typy kontrolerů: *Player* a *Processor*. *Player* a *Processor* jsou konstruovány pro jednotlivé *DataSource* a jsou normálně znovu nepoužitelný k zobrazení jiných médií.



Obr. 14: JMF kontrolery.

3.10.1 Prostředky pro přehrávání videa v JMF

Přímo pro přehrávání časově závislých medií je v jazyce JAVA určeno rozhraní ***Player***, které má metody splňující naše kladené požadavky na přehrávání. Těmito nezbytnými metodami jsou:

- public void **setMediaTime**()

  Nastaví pozici ve video záznamu na zvolený časový údaj.

- public float **setRate**(float factor)

  Nastavuje rychlost přehrávání.

- public void **start**()

  Přesun do stavu *Started*, ve kterém se prezentuje video záznam.

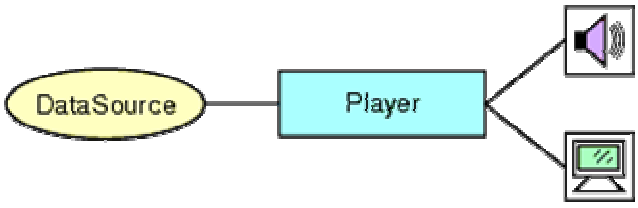
- public java.awt.Component **getVisualComponent**()

  Vrací komponentu, ve které se zobrazuje přehrávaný video záznam.

- public void **stop**()

  Zastaví přehrávání a způsobí do stavu *Prefetched.*

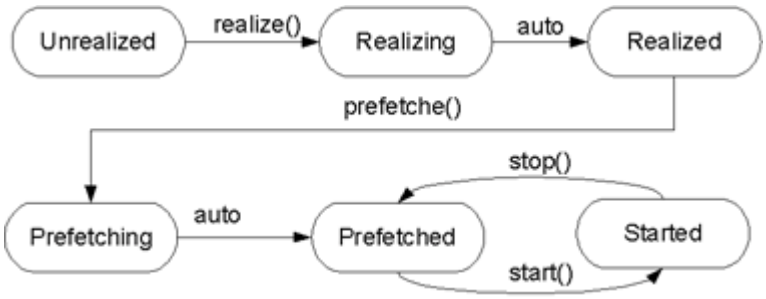
Player zpracovává nějaký multimediální vstupní stream a zobrazuje jej. *DataSource* je používán pro zpřístupnění vstupních dat pro *Player*. Renderovací destinace je závislá na typu vstupního média – např. zvuk a video.



Obr. 15: JMF Player model.

.

Nezbytné pro pochopení fungování všech instancí implementující rozhraní je nutné nastínit, že rozhraní *Player***,** ať už je implementováno jakkoliv, má několik možných stavů. Stavy a přechody mezi nimi ukazuje následující diagram.



Obr. 16: Stavy Playeru

V každém stavu nabízí jiné možnosti manipulace. Metody „realized“ a „prefetched“ jsou zděděny z rozhraní *Controller*. Přechody do stabilních stavů jsou iniciovány metodami k tomu určenými. Tyto metody jsou neblokující a proto po zavolání například metody „realize“ po jejím skončení se nemusí přehrávač nacházet ve stavu *Realized* nýbrž pouze ve stavu *Realizing.* Pro zjištění aktuálního stavu přehrávače slouží metoda „getState“, která je také z rozhraní *Controller*. Při přechodu z jednoho stavu do druhého je zaregistrovanému posluchači typu *ControllerListener* zaslána informace o příslušné změně stavu.

Rozhraní *Player* samozřejmě vytvořit instanci sama sebe neumí, proto je nutné použít třídu *Manager*, která má již prostředky, jak vrátit instanci rozhraní *Player*. Třída *Manager* má všechny metody statické a je chápána jako singleton, proto není možné založit její instanci, která stejně není zapotřebí. Pro pouhé přehrávání bez aplikování různých filtrů a podobných úprav nám bude stačit její přetížená metoda „createRealizedPlayer“, která je připravena v těchto třech verzích:

- public static Player **createRealizedPlayer**(MediaLocator ml)
- public static Player **createRealizedPlayer**(DataSource ds)
- public static Player **createRealizedPlayer**(java.net.URL url)

Metoda *createRealizedPlayer* je blokující a po jejím skončení vrací přehrávač, který je ve stavu *Realized*. Během zpracování požadavku může metoda propagovat několik výjimek, které podávají přesné informace, v čem je problém. Výjimky mohou nastat tyto tři:

- **NoPlayerException**

  Pokud nelze vytvořit přehrávač, je tato výjimka vyvolána.
- **CannotRealizeException**

  Výjimka je vyvolána, pokud přehrávač nelze uvést do stavu *Realized.*
- **java.io.IOException**

  Výjimku vyvolá nedostupnost zdroje dat.

3.10.2 Praktické zkušenosti s přehrávačem

Bohužel praktická zkušenost s těmito výjimkami není dobrá. K vyvolání výjimky občas nedochází a tedy není dost dobře možné na vzniklé situace reagovat. Ani po bližším seznámením s problémem se mi nepodařilo zjistit, proč dochází k výpisu chybové hlášky na standardní chybový výstup a výjimka není vyvolána. Testy jsem prováděl jak na nepodporovaných formátech, tak i na neexistujících souborech. Naštěstí se během implementace můžeme spolehnout na to, že pokud došlo k chybě, tak se nevytvoří žádný přehrávač a do proměnné bude uložena hodnota null. Tomuto neduhu jsem musel upravit i návrh třídy, která se bude starat o přehrávání audio/video záznamu. Je hned několik věcí, které se musí bohužel ošetřovat nepřímo. Proto by se mělo otestovat libovolným způsobem, zda soubor, který se má přehrát, opravdu existuje. A po skončení metody *createRealizedPlayer* provést test, jestli opravdu vytvořila kýžený přehrávač.

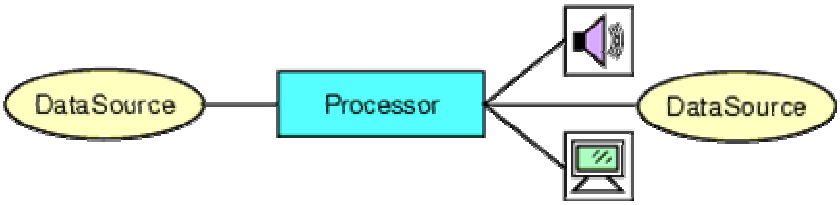
Pokud vše proběhlo bez problémů a kýžený přehrávač je vytvořen, je již velice snadné začít přehrávat data ze zvoleného zdroje. Stačí zavolat metodu *start*, která postupně inicializuje přechody ze stavu *Realized* do stavu *Started*. Dalším problematickým faktorem je, že předem není možné určit, kdy se opravdu zvolená data začnou prezentovat.

Nachází-li se přehrávač ve stavu *Started* je bez problémů možné volání metod *setRate* a *setMediaTime*, ty se už postarají o potřebné akce, které je nutné provést. U metody *setMediaTime* nelze předem určit, jak dlouho jí bude trvat než provede potřebné akce k přesunu v toku dat. Samozřejmě je tato metoda také neblokující. Metoda *setRate* má další specifikum. Tím specifikem je, že se pouze pokusí nastavit požadovanou rychlost přehrávání a zároveň jako návratovou hodnotu vrátí skutečně nastavenou rychlost přehrávání. Tyto dvě rychlosti se vůbec nemusí shodovat.

Bohužel i zde vyvstala chyba. I když se metoda tváří, že nastavila vyšší rychlost přehrávání, opak je pravdou. Testováním bylo zjištěno, že při rychlosti vyšší než 2 se zrychlí akorát rychlost posouvání zobrazovaného ovládacího prvku, který určuje pozici ve videozáznamu. Video se ovšem pořád přehrává blíže neurčenou rychlostí (nejspíše rychlostí 1), která rozhodně neodpovídá rychlosti zvolené. Směrem ke zpomalení je situace lepší, testované zpomalení 0,1 bylo v pořádku.

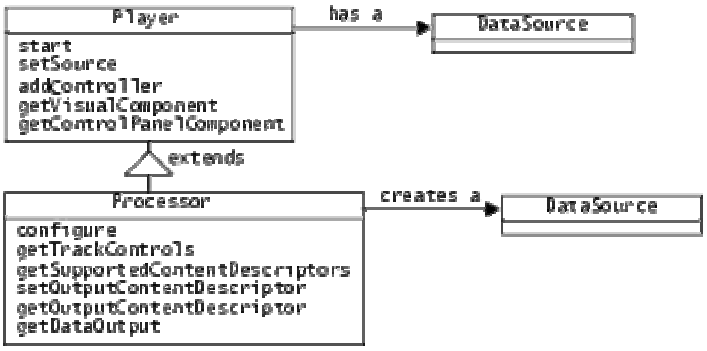
3.10.3 Prostředky pro editaci videa a audia

Procesory mohou být také používány k zobrazení dat. *Processor* je pouze specializovaný typ *Player*, který poskytuje kontrolu nad zpracováním, které je provedeno na vstupu. Procesor podporuje všechny stejné zobrazovací ovládací prvky jako *Player*.



Obr. 17: JMF processor model.

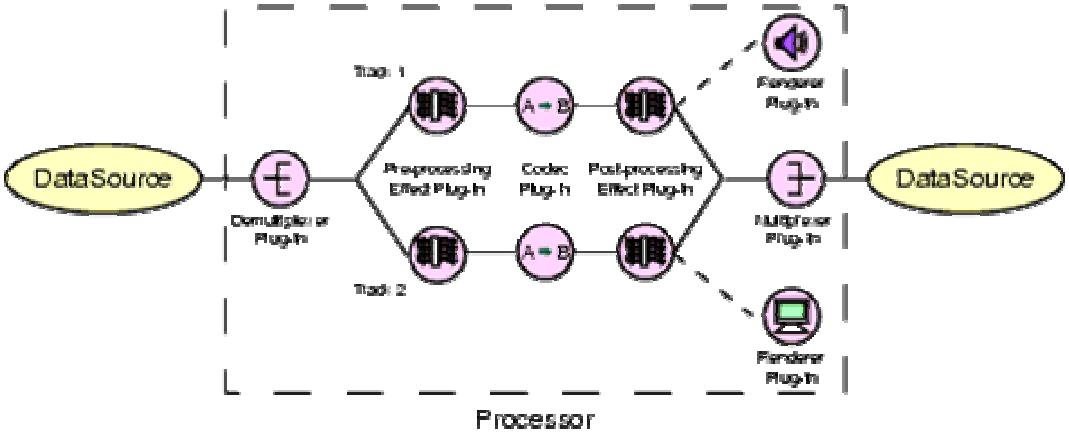
Kromě renderingu u zobrazovacích zařízení, procesor může mít výstup dat přes DataSource tak, že mohou být zobrazeny jiným *Playerem* nebo procesorem, dále zpracovány dalším procesorem, nebo doručeny na některou další destinaci, například soubor.
Procesor je *Player*, který vezme *DataSource* jako vstup, vykoná nějaké uživatelsky definované procesy na datech, a pak má na výstupu zpracovávaná data.



Obr. 18: JMF processors.

Processor může poslat výstupní data do zobrazovacího zařízení nebo do *DataSource*. Jestli jsou data poslána do *DataSource*, tak *DataSource* může být používaný jako vstup jiného *Playeru* nebo *Processoru*, nebo jako vstup do *DataSinku*.

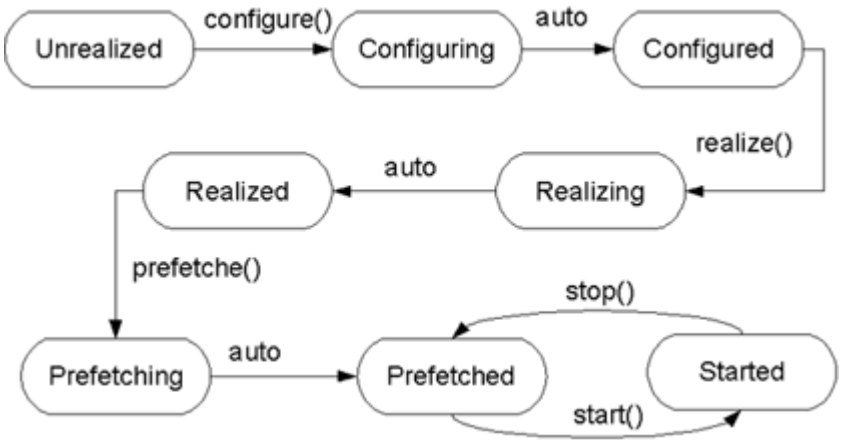
Zatímco se provádí zpracování *Playerem*, *Processor* dovolí vývojáři definovat typ zpracování, které je aplikováno na média. To umožňuje použití efektů, mixování, a kompozice v reálném čase.
Zpracování multimediálních dat lze rozčlenit na několik stupňů:



Obr. 19: Stavy Procesorů.

- **Demultiplexing** je proces rozdělení vstupního toku. Jestliže tok obsahuje mnohonásobné stopy, jsou extrahovány jako samostatné výstupy . Například, QuickTime soubor by mohl být demultiplexován do oddělených audio a video stop. Demultiplexing je provedený automaticky pokud vstupní tok obsahuje multiplexovaná data.
- **Preprocessing** je proces použití výsledných algoritmů na stopy extrahované ze vstupního toku.
- **Transcoding** je proces převádění každé stopy média data z jednoho vstupního formátu na jiný. Když tok dat je převedený z komprimovaného typu na dekomprimovaný typ, to je zpravidla označováno jako dekódování. Naopak, převedení z dekomprimovaného typu na komprimovaný typ je označováno jako kódovaní.
- **Postprocessing** je proces použití výsledných algoritmů na dekódované stopy.
- **Multiplexing** je proces prokládání stop do jediného výstupního toku. Například, oddělené audio a obrazové stopy mohl by být multiplexovány do jediného MPEG-1 datového toku. Můžete specifikovat typ dat výstupního toku s procesorem *setOutputContentDescriptor* metodou.
- **Rendering** je proces zobrazení média uživateli.

Rozhraní *Processor* přidává k rozhraní *Player* několik důležitých funkcí, které umožňují lepší kontrolu nad zpracovávanými daty. Rozhraní *Processor* se ale neliší jen v počtu poskytovaných funkcí, ale i počtem stavů. Do rozhraní *Processor* jsou přidány dva další stavy, které byly přidány kvůli kýžené konfigurovatelnosti. Stavový diagram tedy vypadá takto:



Obr. 20: Stavy Processoru

Než *Processor* dosáhne stavu C*onfigured*, nelze u něj provádět žádná nastavení. Do stavu *Configured* se *Processor* uvede zavoláním metody *configure*. Metoda je také neblokující, proto je nutné ošetřit čekání. Během stavu *Configuring* se může *Processor* pokusit přistoupit a číst ze vstupního souboru či jiného zdroje dat. Jako vstupní zdroj dat mu může sloužit mikrofon, webová kamera nebo jiné podobné zařízení. Jakmile *Processor* dosáhne stavu *Configured*, je zaregistrovanému posluchači zaslána zpráva *ConfigureCompleteEvent*.

3.10.3.1 Metody rozhraní Processor

- void **configure()**

  Metoda je neblokující a uvede **Processor** do stavu *Realizing.*

- *TrackControl*[] **getTrackControls**()

  Metoda vrátí pro každou stopu v datech, instanci třídy *TrackControl***.** Třída *TrackControl* umožňuje různá nastavení stopy.

- ContentDescriptor[] **getSupportedContentDescriptors**()

  Vrátí seznam všech možných výstupů, které *Processor* umí.

- ContentDescriptor setContentDescriptor(ContentDescriptor output)

  Určuje obsah výstupních dat na výstupu.

- DataSource **getDataOutput**()throws NotRealizedError

  Vrací zdroj dat, který bude obsahovat data zpracovaná *Processorem* podle nastavení.

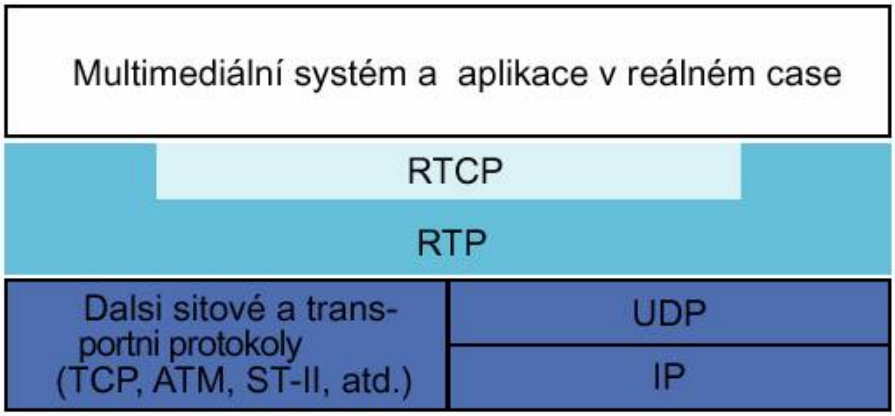
Processor stejně jako *Player*, nemůže založit svoji vlastní instanci, protože se jedná o rozhraní. K vytvoření *Processoru* je tedy nutné použít třídu *Manager*. Třída Manager pro vytvoření *Procesoru* nabízí tyto metody:

- public static Processor **createProcessor**(MediaLocator ml)
- public static Processor **createProcessor**(DataSource ds)
- public static Processor **createProcessor**(java.net.URL url)

V praxi lze řadit několik *Processorů* za sebou, čímž je možné aplikovat několik různých efektů. *Processor* lze použít i na převod na jiný formát. Řazené *Procesory* za sebou spolupracují metodou producent-konzument.

4 Real - Time Transport Protocol (RTP)

Internetovým standardem pro přenos dat v reálném čase je protokol RTP, který obvykle využívá právě protokol UDP z transportní vrstvy, ale může využívat i jiný libovolný protokol s nižší vrstvy.



Obr. 21: Architektura RTP

Protokol RTP může být použit jak pro unicastové (pro každý cíl je vyslána ze zdroje jedna kopie) tak i pro multicastové (data jsou vyslána ze zdroje pouze jednou a je na síti, aby zajistila přenos dat do rozdílných míst) služby sítě. Nezaručuje sice doručení jednotlivých datových paketů ani jejich správné pořadí (to záleží na momentálních možnostech sítě), ale pomocí informací v záhlaví může zjistit jejich pořadí a detekovat ztracené pakety. K zajištění včasného doručení datových paketů nebo k poskytnutí jiných kvalit služeb (QoS) se používá protokol RTCP

4.1 Služby RTP

RTP umožňuje před samotným zahájením přenosu identifikovat typ dat, která budou přenášena, určit pořadí paketu, v jakém budou data zasílána a synchronizovat datové toky z rozdílných zdrojů.

Pro datové pakety RTP není garantováno, ze dorazí v pořadí, v jakém byly vysílány, ani není garantováno, že dorazí všechny. Je na adresátovi, aby rekonstruoval pořadí přijatých paketů a detekoval nepřijaté pakety pomocí informací v záhlaví paketu.

Zatímco RTP neposkytuje žádný mechanismus k zajištění včasného doručení nebo k poskytnutí záruky jiné kvality služeb (QoS), jsou tyto mechanismy poskytovány kontrolním protokolem (RTCP), který umožňuje sledování kvality distribuce dat. RTCP také poskytuje kontrolní a identifikační mechanismus pro přenosy RTP.

4.2 Architektura RTP

Vytvoření RTP spoje je asociace skupiny aplikací, komunikujících s RTP. Spoj je identifikován sítovou adresou a párem portů. Jeden port je určen pro přenos dat a druhý port je určen pro RTCP data.

Účastníkem je jeden stroj, hostitel nebo uživatel účastnící se spojení. Účastí ve spojení může být jednak pasivní příjem dat, vysílání dat nebo dokonce obojí, tj. příjem i vysílání.

Každý rozdílný typ dat je přenášen jiným spojem. Například, pokud je při videokonferenci přenášen zvuk i obraz zároveň, je jeden spoj určen pro přenos audio dat a druhý spoj pro přenos video dat. To umožňuje účastníkovi výběr typu dat, který chce přijímat, např. pokud je někdo v místě s nízkou šířkou pásma, může zvolit pouze příjem audio dat z konference.

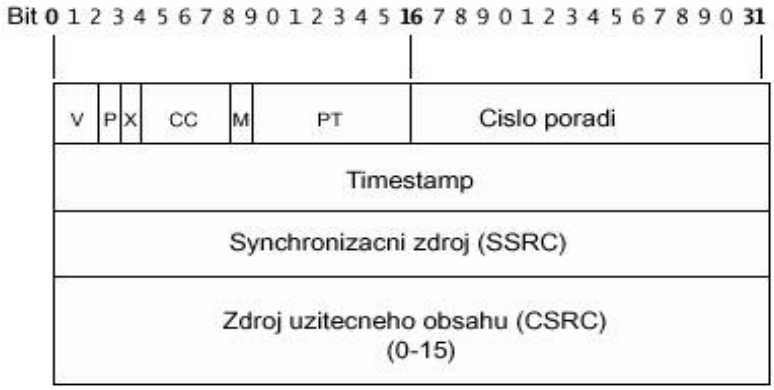
4.3 Datový paket

Záhlaví datového paketu RTP obsahuje:

- Verze RTP protokolu (V): 2 bity
- Doplnění (P): 1 bit. Pokud je tento bit nastaven, je na konci paketu jeden nebo více bytů, které nejsou součástí užitečného obsahu, aby bylo vhodné uspořádání pro případné zabezpečení. Úplně poslední byte v paketu indikuje počet doplněných bytů. Doplnění je využíváno některými šifrovacími algoritmy.
- Rozšíření (X) : 1 bit. Jestliže je tento bit nastaven, následuje za pevným záhlavím jedno rozšíření záhlaví. Tento mechanismus umožňuje vložení rozšiřujících informací do záhlaví RTP.
- Počet CSRC (CC) : 4 bity. (Pole CSRC (Zdroj užitečného obsahu) obsahuje seznam všech SSRC (Synchronizační zdroj), které přispívají nějakým obsahem do paketu. Používá se v případě mixerů, které slučují mediální toky z několika zdrojů.). Počet CSRC identifikátorů, které následují za pevným záhlavím. Jestliže je počet SCRC nula, je zdrojem synchronizace zdroj užitečného obsahu.
- Záložka (M) : 1 bit. Záložkový bit, definovaný konkrétním profilem média. Slouží při rekonstrukci a přehrávání. V případě videa označuje, že je tento paket poslední pro sestavení celého snímku.
- Typ užitečného obsahu (PT) : 7 bitů. Index z tabulky profilu média, který popisuje formát užitečného obsahu (např. JPEG, G.722, H.323).
- Číslo pořadí (SN) : 16 bitů. Číslo paketu, které popisuje pozici paketu v pořadí paketů. Číslo paketu je inkrementováno po každém odeslaném paketu. Používá se pro zjištění ztrát nebo duplicity paketů a obnovení původního pořadí paketů.

- TIMESTAMP: 32 bitů. Vyjadřuje moment odebrání vzorku prvního bytu užitečného obsahu. Označuje, kdy byl rámec pořízen a vygenerován
- SSRC : 32 bitů. Díky němu se snadno rozliší zda jednotlivé toky pochází z jednoho mixeru nebo z translátoru a identifikuje se synchronizační zdroj na základě zpráv přijímačů. Jestliže je počet CSRC roven nule, je zdroj užitečného obsahu zdrojem synchronizace. Jestliže je CSRC nenulové, SSRC identifikuje mixér.
- CSRC : 32 bitů každý. Identifikuje zdroje přispívající do užitečného obsahu. Počet přispívajících zdrojů je určen polem počtu CSRC (CC). Celkem může být 16 přispívajících zdrojů. Jestliže je více přispívajících zdrojů, je výsledný užitečný obsah sloučením těchto zdrojů.

Datový paket protokolu RTP je na obrázku 22.



Obr. 22: Formát záhlaví datového paketu RTP

5 Real Time Control Protocol (RTCP)

Protokol RTCP je řídícím protokolem RTP, pomocí něhož je monitorováno doručování paketů. Protože protokol RTP neposkytuje žádný mechanizmus na zajištění doručení, včasného doručení paketů, ani doručení paketů ve správném pořadí. Tyto dva protokoly jsou často brány dohromady a označovány jako RTP/RTCP.

RTCP používá UDP port o jedničku vyšší než používá RTP. RTCP vytváří zpětnou vazbu mezi účastníky relace protokolu RTP, ve které periodicky probíhá výměna RTCP paketů. RTCP pakety obsahují informace, podle kterých může strana vysílající multimediální proud dynamicky měnit např. rychlost přenosu na základě požadavků přijímající strany. Protokol RTCP tak poskytuje služby řízení toku a kontroly zahlcení sítě.

První paket ve skupině RTCP paketů musí být paket zprávy, i v případě, že nebyla přijata či odeslána žádná data. Zpráva vysílače popisuje celkové množství vyslaných dat, obsahující údaje o absolutním čase, aby umožnily vzájemnou synchronizaci médií. Zpráva příjemce obsahuje informaci o počtu ztracených paketů, nejvyšší číslo pořadí a TIMESTAMPu, jenž může být užit k odhadu obousměrného zpoždění mezi odesílatelem a příjemcem. Informace zpráv lze využít k řízení datového toku pro vysílač, k lokalizaci chyby pro příjemce, zda jde o chybu lokální, regionální či globální. Síťový administrátor může tyto informace využít pro vylepšení výkonu distribuce multicastových dat.

RTCP zajišťuje několik následujících funkcí:

- Zajišťuje předávání informací aplikaci. Jde o hlavní funkci RTCP. Experimenty s multicastovou technologií ukazují, že je důležité mít zpětnou vazbu z RTCP, aby bylo možno zjišťovat, zda nenastávají problémy při distribuci dat. Každý RTCP paket obsahuje zprávy od vysílače/přijímače.

- Identifikuje zdroj RTP. Všechny skupiny RTCP paketů obsahují popis zdroje (SDES), který obsahuje kanonické jméno (CNAME), jenž identifikuje zdroj a využívá se pro synchronizaci audia a videa na straně příjemce, neboť distribuce těchto dvou složek probíhá odděleně, každá na svém vlastním RTP potu. CNAME je jedinečný identifikátor ve formě podobné e-mailové adrese. Používá se také pro řešení konfliktů s hodnotou SSRC a k přiřazení různých mediálních toků generovaných stejným uživatelem.

- Kontroluje si vlastní přenos svých paketů. RTCP protokol si sám kontroluje přenos svých paketů a hlídá, aby nedocházelo k jejich nadměrnému vysílání. Horní hranice je 5% z celkového přenosu jednoho sezení. Informace je distribuovaná a každý účastník sezení ví o ostatních účastnících a na základě toho si určuje poměr pro vysílání RTCP paketů.

- Zajišťuje minimální distribuci kontrolních informací. Jde o volitelnou funkci RTCP. Jde o zajištění přenosu informací ke všem účastníkům. Pakety SDES identifikují například jméno účastníka, e-mailovou adresu a telefonní číslo. Poskytují tak jednoduché řízení sezení. Klientské aplikace pak mohou na grafickém uživatelském prostředí zobrazit tyto osobní informace. Účastníci se tak dozví o dalších účastnících v sezení, získávají kontaktní informace a podmíní další formy komunikace.

Pokud již není zdroj aktivní, vyšle protokol RTCP paket BYE. Toto oznámení může obsahovat důvod, proč zdroj opouští spoj.

6 MULTICAST

6.1 Co je multicast

Multicast je mechanismus routování (směrování) IP paketů, který zajistí (na tzv. principu "best effort"), že paket vyslaný odesílatelem bude doručen každému zařízení z dané cílové skupiny. Této skupině se říká multicastová skupina.

V rámci jedné multicastové skupiny lze např. přijímat video vysílané do této skupiny streaming serverem. Aby mohl uzel přijímat data poslaná do multicastové skupiny, musí být jejím členem. Hlavním cílem této technologie je zásadní odlehčení zátěže vysílajícího uzlu a přenosové soustavy při přenosech typu jeden zdroj - mnoho příjemců. Zdroj tedy vysílá data, určená neznámému, potenciálně velmi velkému počtu příjemců pouze jednou a veškerá režie spojená s distribucí příjemcům je ponechána na přenosové soustavě, v prostředí Internetu tedy (v ideálním stavu) na routerech (směrovačích). Na nich také je, aby zajistily efektivní přenos dat od zdroje k příjemcům, tedy aby vysílaná data poslaly po každém spoji nejvýše jedenkrát, a to pouze tehdy, je-li daným směrem skutečně nějaký příjemce. Na rozdíl od klasického přímého vysílání (*unicast*), kdy přenos paketu dat od zdroje k cíli je iniciován zdrojem, je tok paketů skupinového vysílání určován příjemci.

Multicast je integrální součástí protokolu IP (Internet Protocol). Internet Protocol Multicast je technologie, která slouží k efektivnímu využití šířky pásma pro situace, kdy jeden vysílač distribuuje informaci (v praxi např. videokonference, software, burzovní zprávy atd.) tisícům příjemců.

6.2 Adresy pro multicast

K identifikaci skupin příjemců se používá speciální třída IP adres (třída D), zahrnující adresy z množiny adresy 224.0.0.0 až 239.255.255.255. První čtyři bity (1110) identifikují multicastový paket, zbytek (28 bitů) říká, pro kterou multicastovou skupinu je daný uzel určen.

Vysílající uzel odesílá pakety dat s cílovou adresou skupiny (a svou vlastní obyčejnou zdrojovou adresou). Další šíření přes směrovače by mělo probíhat stejnou metodou „best effort“ jako šíření běžných paketů přímého vysílání. V případě skupinového vysílání ovšem může směrovač provést replikaci paketu a jeho vyslání do více směrů.

Organizace IANA rozdělila adresní prostor pro multicast následovně: 224.0.0.0 - 224.0.0.255 jsou určené pro lokální použití, např. 224.0.0.5 je adresa všech lokálních OSPF routerů. Pro globální použití jsou pak určeny adresy z rozsahu 224.0.1.0 až 238.255.255.255. Adresu z tohoto rozsahu budou využívat i mix servery, protože mohou být rozmístěny po celém světě. Některé adresy z tohoto

rozsahu už byly rezervovány pro specifické použití, takže je potřeba vybrat nějakou, která nekoliduje se stávajícími. Existují ještě další rozsahy z třídy D, které mají pro multicast zvláštní sémantiku.

6.3 Skupinové vysílání v lokální síti

Protokoly na 2. vrstvě síťové hierarchie (v našich podmínkách je z nich nejrozšířenější ethernet) obsahují ve svých specifikacích podporu skupinového vysílání v podobě speciálních MAC adres. Za normálních okolností přijímá síťová karta (NIC - Network Interface Card) pouze rámce určené pro jeho MAC adresu nebo pro broadcast adresu. Běžné síťové karty pracovních stanic (včetně PC) pak mají schopnost podle svého okamžitého nastavení (na základě požadavků programu) filtrovat pakety skupinového vysílání a nejbližším vrstvám programového vybavení již předávat jen relevantní část paketů skupinového vysílání, které se v lokální síti pohybují, tedy pouze skupiny, jež jsou předmětem momentálního zájmu dané stanice. Nedochází tedy k zatěžování stanic lokální sítě, jichž se dané skupinové vysílání netýká.

6.4 Přenos skupinového vysílání mezi sítěmi, protokol IGMP

Snadnost implementace skupinového vysílání v rámci lokální sítě se vytrácí, jakmile chceme dosáhnout přenosu v rámci propojených sítí. Nejdůležitějším článkem jsou směrovače s jejich primárním úkolem získat informace o tom, které skupiny mají být vysílány do sítí, jež jsou ke směrovači bezprostředně připojeny. K tomuto účelu byl vyvinut speciální protokol IGMP (Internet Group Management Protocol).

IGMP slouží k dynamické registraci uzlů do multicastové skupiny. Jeho pomocí tedy směrovač periodicky zjišťuje zájem stanic v připojených sítích o jednotlivé proudy skupinového vysílání. Směrovač vyšle do připojené sítě dotaz (paket se speciální skupinovou adresou 224.0.0.1) a jednotlivé stanice odpovídají (s náhodně zvoleným zpožděním, aby nedocházelo k zahlcení sítě při současné odpovědi všech najednou) informací o adresách skupinového vysílání, o něž mají zájem. Odpovědi jsou rovněž vysílány na adresu 224.0.0.1 a odposlouchávány ostatními stanicemi. Tím se zamezí duplicitnímu vysílání požadavků na stejnou skupinu. Programové vybavení koncové stanice tedy musí navíc podporovat protokol IGMP. Směrovače tak pomocí protokolu IGMP sledují zájem o příjem konkrétních skupin ve svém bezprostředním okolí.

6.5 Multicast routing

Běžné směrovače v Internetu nejsou zkonfigurovány tak, aby předávaly zprávy pro multicastové skupiny. Směrovače, které jsou schopny přenášet multicastové pakety tvoří distribuční síť ve tvaru stromu. Existují dva typy distribučních stromů - zdrojové (source) a sdílené (shared) stromy.

Jednodušší je zdrojový strom, kde kořen stromu je reprezentován vysílajícím uzlem a jeho větve tvoří (minimální) kostru skrz síť. Sdílené stromy používají společný kořen, který je umístěn ve zvoleném místě sítě. Tento sdílený kořen je nazýván rendezvous point (RP). Když se používá sdílený strom, všechny zdroje musí přeposlat síťový provoz na kořenový uzel, odkud je dále distribuován do sítě. Oba typy stromů - zdrojové a sdílené neobsahují kružnice. Zprávy se replikují pouze tam, kde se strom větví.

Členové multicastové skupiny se mohou kdykoliv připojit nebo odhlásit, takže distribuční strom musí být dynamicky updatován. Pokud všichni aktivní členové v určité větvi přestanou vyžadovat síťový provoz pro danou multicastovou skupinu, je tato větev odříznuta směrovačem z distribučního stromu. Pokud se nějaký člen z této větve znovu přihlásí k odběru zpráv pro danou multicastovou skupinu, směrovače dynamicky modifikují distribuční strom a začnou znovu přeposílat síťový provoz pro tuto skupinu po této větvi. Směrovače musejí kromě trvalého mapování svého bezprostředního okolí zajistit tok paketů skupinového vysílání i do vzdálených oblastí sítě, a to pokud možno optimálním způsobem. K tomu slouží tzv. směrovací protokoly. Jejich pomocí směrovače hledají minimální strom spojů pokrývající cestu od zdroje skupinového vysílání k momentálním zájemcům o příjem. Je zřejmé, že na rozdíl od klasického směrování přímého vysílání půjde o proces velmi dynamický. Cesta od daného zdroje k danému cíli je totiž stálá, pokud nedojde k nějaké vnější události měnící topologii sítě, např. poruše linky. Naproti tomu zájemci o příjem daného skupinového vysílání mohou vznikat a zanikat trvale a tento proces průběžných změn musejí směrovací protokoly vhodně reflektovat. Směrovací protokoly skupinového vysílání jsou dosud předmětem intenzivního výzkumu a vývoje. V současné době se nejvíce používají protokoly DVMRP (Distance Vector Multicast Routing Protocol) a dvě varianty protokolu PIM (Protocol Independent Multicast).

7 Architektura aplikace

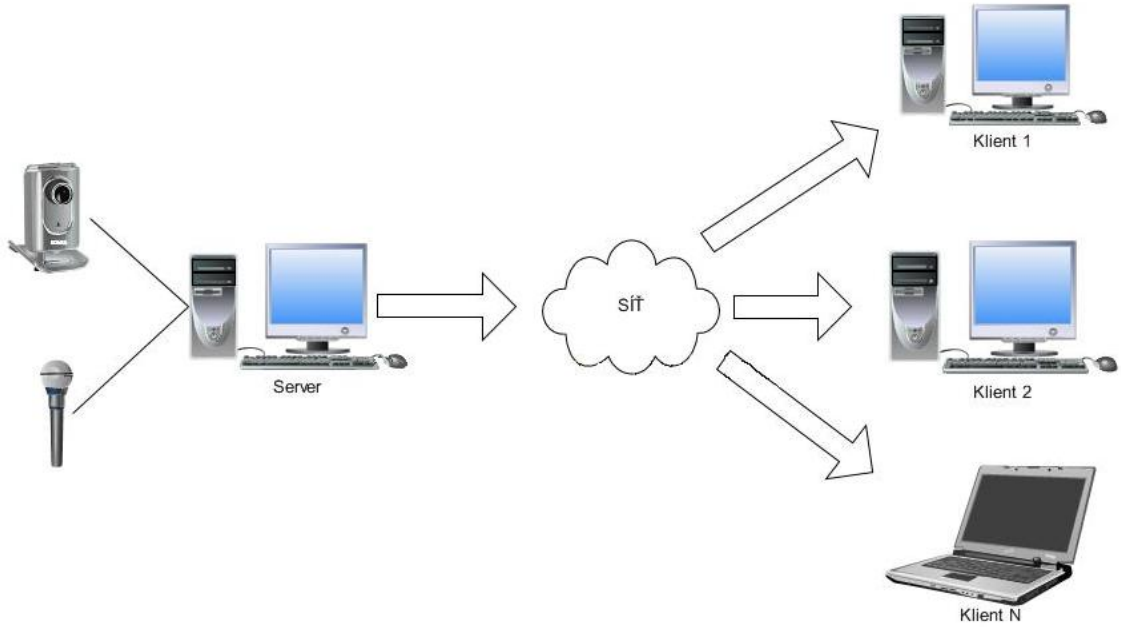
Jak již bylo řečeno v úvodu, cílem bylo vytvořit dva programy z nichž jeden, který je nazván Server, bude umístěn na počítači, který obsluhuje přednášející. Druhý program, který je nazván Klient a bude na počítači, kde bude student nebo někdo, kdo má zájem o danou přednášku. Server se nachází na počítači, ze kterého jsou snímána data a vysílána do sítě. Tyto aplikace, tedy Server a Klient, může využívat kdokoliv a nemusí být vždy využity pro přednášku. Mohly by být například využity ke sledování nějakého pokusu a určitě by se našla i jiná využití. Aplikace byla navrhnuta a naprogramována tak, aby byla přenositelná na jakýkoliv počítač s připojením k internetu nebo síti LAN.

K návrhu aplikace jsem vycházel z předpokladů uvedených v úvodu. Čili máme dvě nezávislé aplikace: Server a Klient. Na serveru běží veškerá logika související s během videokonference, jako je správa uživatelů, funkce chatu a její zprostředkovávání klientům, atd. Klient funguje jako přijímač vysílaného streamu serverem a zobrazuje zprávy posílané všemi uživateli konference. Další jeho neméně důležitou funkcí je vysílání obrazu uživatele k serveru a pokud chce přednášející, tak i ostatním klientům.

Aplikaci jsem implementoval tak, aby se dala spustit a byla přenositelná na jakýkoliv počítač připojený k internetu nebo síti LAN. Aplikaci lze vyzkoušet i na jednom stroji, bez nutnosti sítě. Pro implementaci přenosů dat mezi klientem a serverem jsem použil TCP/IP protokol implementovaný pomocí soketů. Sokety jsem si vybral z důvodu jednoduchosti, žádného nastavovaní a všeobecné použitelnosti. Mezi akceptovatelná řešení patří i RMI. RMI je zkratka z anglického Remote Metod Invocation, což je vzdálené volání metod. Je to silný nástroj, který ovšem vyžaduje mnoho nastavení související s během aplikace na pozadí. Tato nastavení by bylo možné realizovat pomocí skriptu, i přesto by byl zásah uživatele nezbytný. Tato problematika je více probrána v kapitole 7.1.

Pro přenos multimedií jsem použil Java Media Framework a protokol RTP. Hlavní logika přenosu se skládá ze dvou částí – vysílače a přijímače. Při použití JMF jsem narazil na velkou škálu problémů z důvodu velké chybovosti frameworku. Poslední verze vyšla v roce 2003 a už se nevyvíjí. Hlavním problémem je velká náročnost na procesor a mnoho chyb, které vedou k chybnému přenosu nebo pádu celé logiky přenosu. Více v kapitole 7.4.

Pro vytvoření GUI jsem použil Java knihovnu Swing.



Obr. 23: Obecný pohled na aplikaci

7.1 Přenos informací mezi serverem a klientem

V poslední době se vývoj aplikací typu klient/server přesunul na vícevrstvé aplikace. Návrh vícevrstvé aplikace je založen na předpokladu, že aplikace je rozdělena na tři součásti neboli „vrstvy“: na prezentační vrstvu (GUI), aplikační logiku (business logic), a na vrstvu zdrojů dat, což je většinou databáze. Často je obvyklé, že aplikační logika a GUI jsou od sebe oddělené tak, že běží na různých počítačích. Jsou-li jednotlivé součásti aplikace fyzicky uloženy na oddělených počítačích, jde o **distribuovanou aplikaci**. Termín distribuovaný objekt znamená, že objekt umožňuje volání svých metod z procesů běžících na jiných počítačích nebo z paměťového prostoru jiného virtuálního stroje jazyka Java. Pro použití distribuovaných objektů jsem volil mezi těmito technologiemi:

- **Sokety**
- **Architektura Corba**
- **Architektura RMI**
- **Objektový model Enterprise JavaBeans**

Pro tvoření distribuovaných systémů v jazyce Java se nejvíce hodí technologie RMI. Architektura RMI poskytuje velice jednoduchý a pohodlný způsob tvorby distribuovaných objektů. Od jejího použití v aplikaci mě odradilo mnoho nastavení a spouštění externích programů pro správný chod aplikace. Toto nastavení můžeme zautomatizovat pomocí skriptů. Ale i toto by se neobešlo bez

zásahu uživatele. Toto vyvrací mojí strategii vytvořit aplikaci, která je přenositelná na kterýkoliv počítač bez žádného nastavení. Proto jsem použil architekturu pomocí soketů.

7.1.1 Implementace pomocí soketů

Sokety poskytují aplikační rozhraní (API), které zajišťuje přímý přístup k síťovým protokolům nižší úrovně a umožňuje použít ke komunikaci dvou procesů spuštěných na dvou samostatných počítačích protokol TCP/IP. Ten poskytuje komunikaci v rámci vzájemně propojených sítí tvořených počítači s různou hardwarovou architekturou a různými operačními systémy. Třídy soketů jsou v Javě definovány v balíčku *java.net*. Chceme-li vytvořit propojení dvou procesů, musí jeden z nich k čekání na příchozí připojení použít instanci třídy *ServerSocket*., zatímco druhý proces musí požadované připojení vytvořit pomocí instance klientské třídy *Socket*.

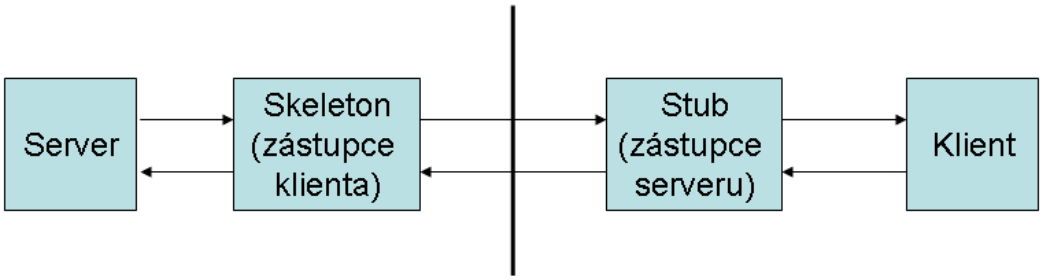
7.1.1.1 Implementace na straně serveru

Nejdůležitější funkčnost na serveru je možnost připojení klientu k tomuto serveru. Kód na straně serveru musí tudíž vytvořit instanci typu *ServerSocket* a čekat na příchozí připojení. Server čeká v nekonečném cyklu na příchozí připojení. Je-li spojení s klientem navázáno, vytvoří instanci typu *Socket.* Aby mohl server podporovat více uživatelů, musí být schopen čekat na informace ze všech připojených uživatelů. Toto jsem vyřešil tak, že jsem pro každého uživatele vytvořil samostatné vlákno. Toto vlákno čeká jen na informace od příslušného uživatele.

```
public class SocketServer {

    Socket client;
    ServerSocket ss = new ServerSocket(PORT_NUMBER);
    while (true) {
        client = ss.accept();
        SocketSkeleton skel = new SocketSkeleton(client);
        addClient(skel);
        skel.start();
    }
  }

  class SocketSkeleton extends Thread {

    protected Socket receiveSocket;
    protected Socket sendSocket;

    public SocketSkeleton(Socket client) {
      receiveSocket = client;
    }

     public void run() {
```

```
            //…
        }

    }
```

Instance vnitřní třídy nazvaná *SocketSkeleton* slouží jako objekt *proxy* (zástupce klienta) na serverovém počítači. Volání metody ze třídy *SocketSkeleton* bude přenášeno po síti do klientského kódu na počítači uživatele. Komunikace mezi klientem a serverem je na obrázku číslo 24.



Obr. 24: Architektura tříd pro přenos informací mezi klientem a serverem

Sokety jsou primitivním komunikačním mechanismem pro posílání a přijímání dat, při němž implicitně neposkytují žádnou možnost, jak z jednoho procesu volat metody jiného procesu. Tuto funkci lze snadno simulovat definicí protokolu, který umožní klientskému kódu odesílat na server taková data, jež jasně specifikují, jakou metodu je třeba volat. Každé metodě na serveru lze například přiřadit celočíselnou hodnotu. Klientský kód pak na server odešle pouze tuto hodnotu. Na jejím základě už server bude dobře vědět, co má udělat.

V následujícím kódu vytvoří metoda *initialize()* dvojici objektů typu *InputStream* a *OutputStream*, kterou použijeme jednak ke čtení příchozích příkazu (volání metod), jednak odesílání výsledků klientovi. Nové vlákno bude čekat na hodnotu typu *int*, kterou porovná s konstantami nadefinovanými na třídě serveru. Bude-li získána hodnota s jednou z konstant souhlasit, spustí server požadovanou metodu. Vyžaduje-li metoda argumenty, budou příslušné hodnoty načteny z instance *InputStream* (z vstupního proudu). Návratové hodnoty budou uloženy do instance typu *OutputStream* (do výstupního proudu) , které se vrací zpět na klienta.

```
public class SocketServer {

  public final static int PORT_NUMBER = 1234;
  public final static int INVOKE_GET_HISTORY = 0;
  public final static int INVOKE_BROADCAST = 1;

 // ...
}
```

```
class SocketSkeleton extends Thread {

//...
    public void run() {
      int methodID;
      try {
        initialize();
        while (true) {
          //identifikace metody podle hodnoty int
          methodID = inInput.readInt();
          switch (methodID) {
            case INVOKE_BROADCAST:
              handleBroadcast();
              break;
            case INVOKE_GET_HISTORY:
              handleGetHistory();
              break;
          }
        }
      } catch (Exception ioe) {}
    }

    protected void handleBroadcast() throws IOException {
      //...
    }

    protected void handleGetHistory() throws IOException {
      // ...
    }

    protected void initialize() throws IOException {
      InputStream is;
      OutputStream os;
      InetAddress addr;

      is = receiveSocket.getInputStream();
      inInput = new DataInputStream(is);
      os = receiveSocket.getOutputStream();
      inOutput = new DataOutputStream(os);

    }

 // ...

}
```

7.1.1.2 Implementace na straně klienta

Veškerá logika pro distribuovaný přenos na straně klienta je ve třídě *SocketClient.* Vzhledem k funkcím, o které se stará server, musí mít klient následující funkce:

- Musí vytvořit připojení k serveru. K tomu použije číslo portu, které server sleduje. Zmíněné připojení bude server používat ke zpracování klientských volání. Tímto kanálem bude navíc rovněž vracet i výsledky volání.
- Musí vytvořit instanci typu *ServerSocket*, kterou použije k vytvoření druhého připojení mezi klientem a serverem. Toto připojení umožní serveru volat metody ze strany klienta.

Stejně jako třída *SocketServer* definovala klienta proxy nazvaného *SocketSkeleton* (zástupce klienta na straně serveru), definuje i třída *SocketClient* server proxy nazvaný *SocketStub*. Tato třída slouží k převodu volání serverových metod na data, která lze posílat prostřednictvím socketového připojení. Server proxy se používá rovněž k načtení výsledků zmiňovaných volání. Jeho metoda *run()* umožňuje vláknu čekat v nekonečném cyklu na data ze serveru. Viz obrázek 24.

7.2 Aplikace Server

Server je aplikace, kterou spouští přednášející osoba. Tato aplikace se stará o veškerou logiku související s funkcí videokonference. Kromě hlavní funkce jako je vysílání audio/video streamu zde běží TCP server, který podle různých příznaků zpracovává data a pracuje s nimi. Pomocí toho serveru funguje chat, správa uživatelů i správa portů. Vše o přenosu informací mezi serverem a klientem je podrobněji popsané v kapitole 7.1.

Aplikace server v sobě obsahuje z určité části obsahu i klienta. A to v souvislosti s chatem a příjmem audio/video streamu od klienta, se kterým se přednášející rozhodl spojit. Logika ohledně přenosu A/V informací je podrobněji popsána v kapitole 7.4.

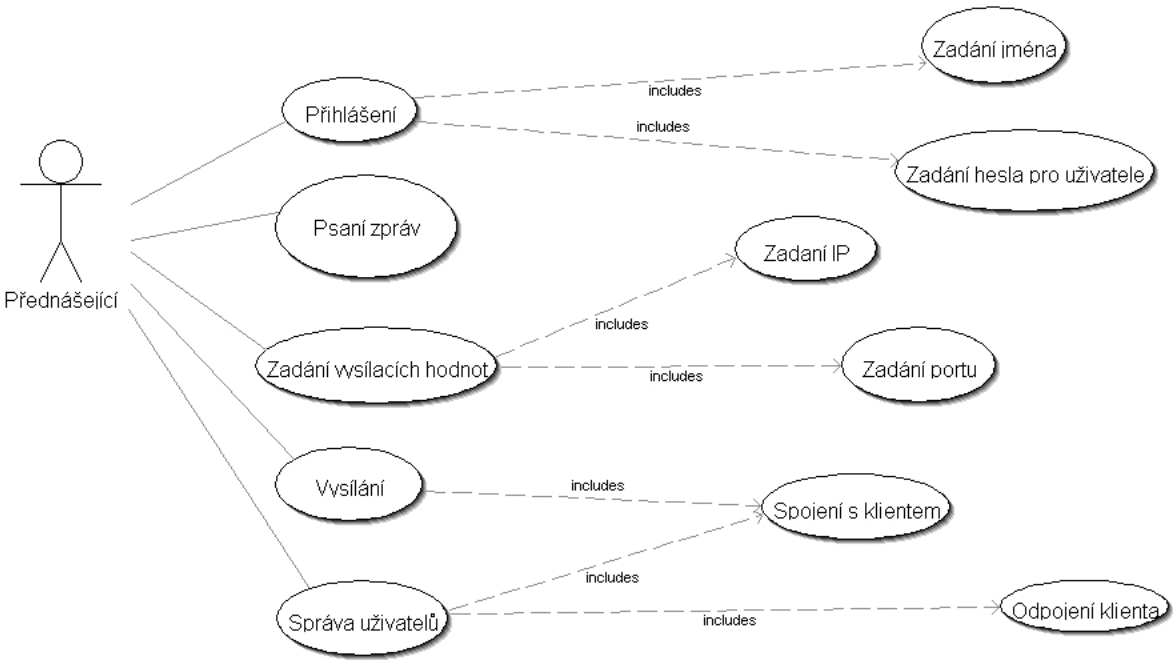
7.2.1 Seznámení s aplikací Server

Přednášející začíná konferenci přihlášením do aplikace. V této fázi zadá své jméno, pod kterým bude vystupovat a zároveň pro bezpečnost zadá heslo pro přihlášení do konference pro klienty. Čili pokud klient nezná heslo zadané přednášejícím, nemůže se přihlásit do aplikace. Toto heslo musí přednášející nějakými informačními kanály distribuovat všem uživatelům, kteří by mohli mít o danou konferenci zájem.

Po přihlášení přednášející zadá IP adresu, pod kterou se bude vysílat, ovšem ta by měla být stejná jako je adresa počítače a port, na kterém se bude vysílat. V nastavení aplikace nastaví vstupní zařízení pro kameru a mikrofon. Poté může začít vysílat. Do té doby dokud nezačne vysílat se nepřipojí žádný uživatel.

Jakmile se do videokonference připojí nějaký klient, přednášející má možnost jej spravovat. Může s ním komunikovat přes chat a má možnost se s ním spojit audio nebo audio/video kanálem. Klient tuto potřebu přijme nebo odmítne. Velice důležitá možnost je, že přednášející chce svůj rozhovor s daným klientům poskytnout i ostatním účastníkům konference. Ti už daný obraz a zvuk začnou přijímat automaticky.

Další možností přednášejícího je odpojení nevhodného účastníka ven z konference.



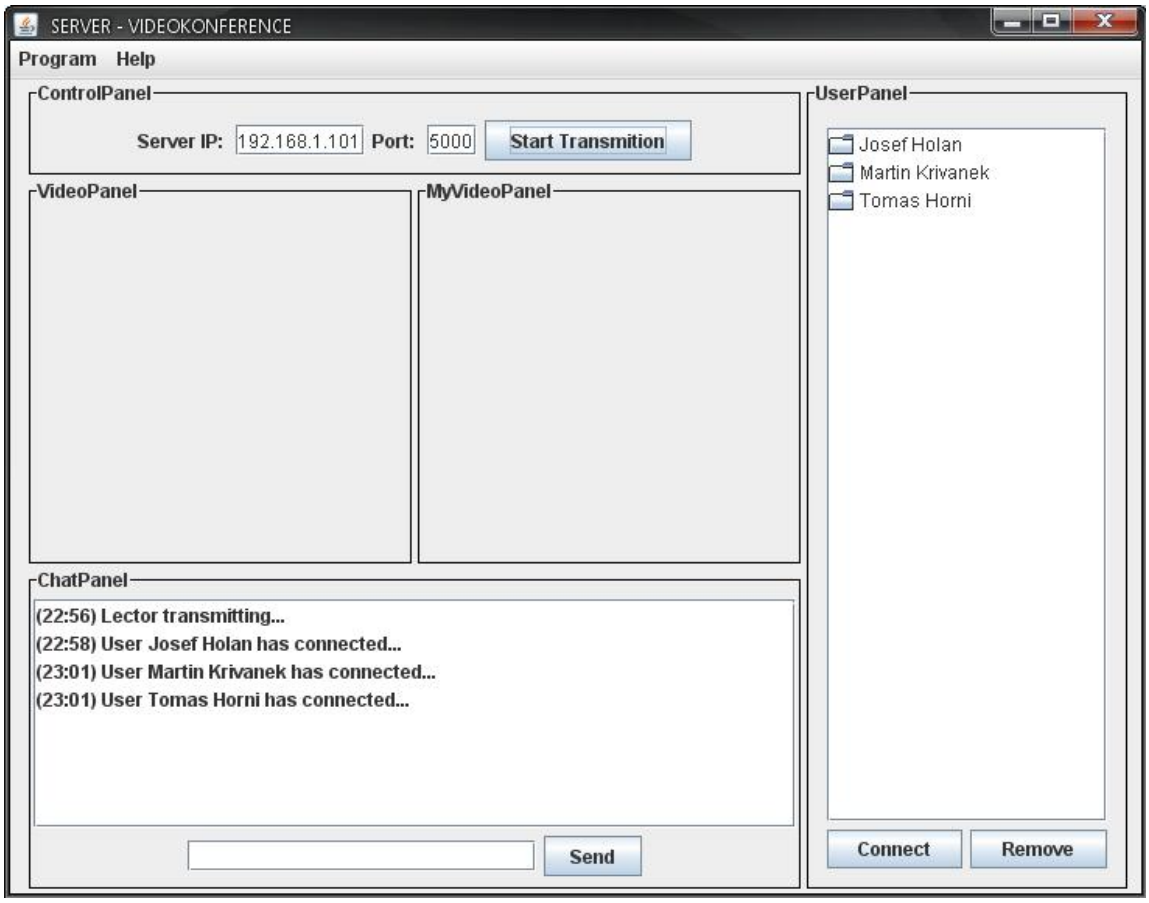
Obr. 25: Diagram použití

7.2.2 Struktura aplikace

Základem celé aplikace je hlavní panel, do kterého se vkládají další komponenty jako je menu a další vnitřní panely. Tento panel je zkonstruován pomocí instance třídy *ServerGUI*. Tato třída rozšiřuje třídu *JFrame* z knihovny Swing.

Pro jednoduchost a přehlednost jsem celou aplikaci rozdělil do několika funkčních oblastí. Každou tuto oblast znázorňuje panel. Z obrázku č. 26 je vidět, že se aplikace skládá z šesti základních panelů :

- **Control Panel**
  Panel pro nastaveni IP adresy a portu, na kterém se bude vysílat.
- **Video Panel**
  Panel, na kterém přednášející uvidí klienta, se kterým se rozhodl spojit.
- **MyVideo Panel**
  Panel, na kterém přednášející vidí a slyší vlastní obraz a zvuk z vlastní kamery a mikrofonu.
- **User Panel**
  Panel pro správu uživatelů. Zde se zobrazuje list připojených uživatelů, ke kterým se přednášející kdykoliv může připojit.
- **ChatPanel**
  Panel pro písemnou komunikaci mezi uživateli. Do tohoto panelu se vypisují i informativní hlášky ze serveru.
- **MenuPanel**
  Panel pro zobrazení menu.

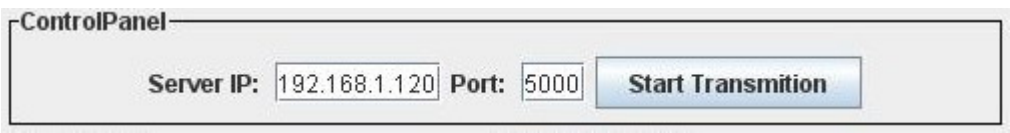


Obr. 26: Vzhled aplikace Server

Pří návrhu aplikace jsem se snažil o znuvupoužitelnost, což mělo za následek zapouzdřenost objektů. Takže ke všem proměnným v daných třídách se musí přistupovat přes *get* a *set* metody.

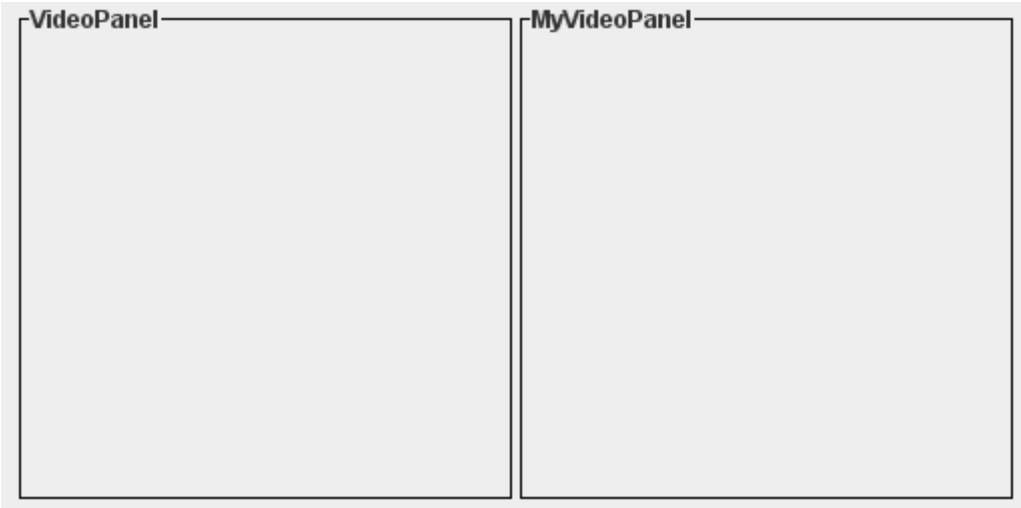
Problém předávání objektů, proměnných a konstant mezi panely jsem vyřešil tak, že při vytváření instancí panelů jim předávám instanci hlavního panelu, což je třída *ServerGUI*. Tato třída zná instance všech panelů a tudíž přes ni může kterýkoliv panel přistupovat k proměnným jiného panelu.

7.2.2.1 Control Panel



Control panel slouží jen pro nastavení portu, na kterém dojde k vysílání. Pomocí tlačítka **Start Transmiton** spouštíme a vypínáme přenos. Ve třídě *ControlPanel* se nenachází žádná funkčnost, tato třída se stará pouze o GUI.

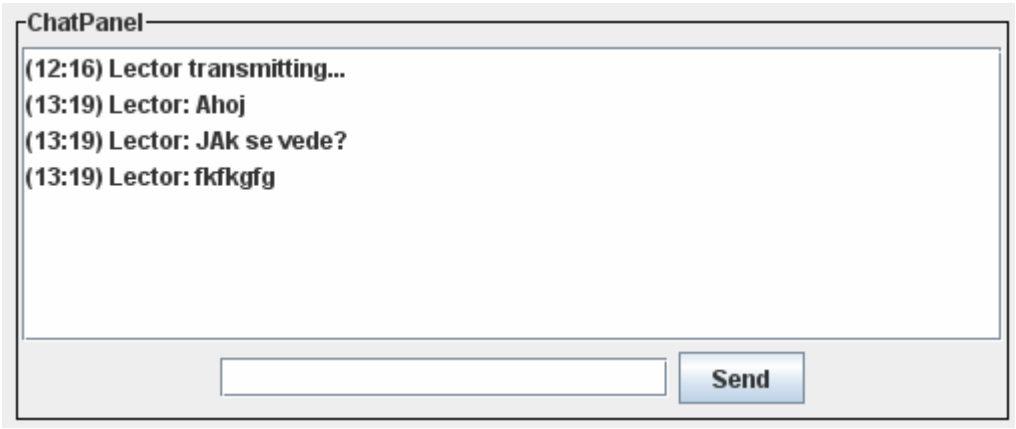
7.2.2.2 VideoPanel a MyVideoPanel



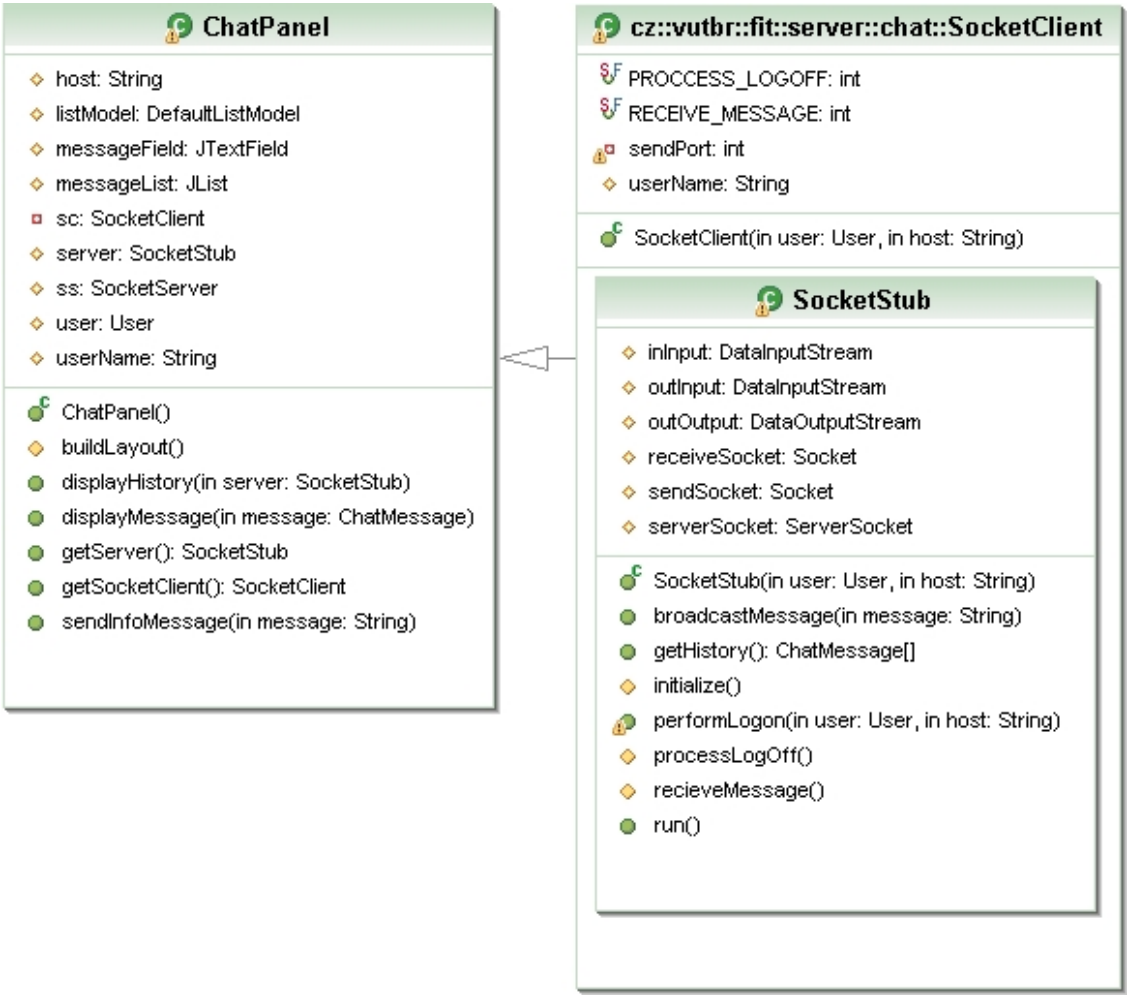
Oba tyto panely slouží jen k zobrazení přenášeného videa. *MyVideoPanel* zobrazuje obraz přenášený z kamery na straně serveru. A *VideoPanel* zobrazuje video přenášené od klienta. Tento panel začne zobrazovat hned jakmile se přenášející rozhodne spojit individuálně s nějakým uživatelem a uživatel odsouhlasí vysílání. Obě třídy těchto panelů obsahují pouze metody pro vytvoření GUI. Ve třídě *MyVideoPanel* se inicializuje Třída *AVReciever*, což je přijímač, který přijímá, zpracovává vysílané video a zvuk. V kontrolní liště se zobrazuje ovládání hlasitosti, a délka přenosu. Více o vysílání o přijímání videa v aplikace je popsáno v kapitole 7.4.

7.2.2.3 ChatPanel

Třída *ChatPanel* umožňuje uživateli posílat zprávy dalším uživatelům a zároveň si tyto zprávy číst. Skládá se ze tří komponent, panelu na kterém se zprávy zobrazují, textového pole, do kterého se zprávy píší a odesílacího tlačítka.



Obr. 27: Detail ChatPanelu



Obr. 28: Detail základních tříd panelu ChatPanel

Třída *ChatPanel* je společná pro aplikace Klient i Server. Stará se jen o zobrazování zpráv od uživatelů konference a správu historie. V pozadí stále běží démon *SocketServer*, který neustále naslouchá a volá potřebné metody jako je *displayMessage()* a *displayHistory()*.

Základem *ChatPanelu* je *JScrollPane*, ve kterém se zobrazuje seznam zpráv:

```
protected JList messageList;
protected DefaultListModel listModel;

listModel = new DefaultListModel();
messageList = new JList(listModel);
JScrollPane scrollPane = new JScrollPane(messageList);
this.add(scrollPane, BorderLayout.CENTER);
```

Metoda *displayMessage(ChatMessage message)* vkládá do proměnné *listModel*, což je seznam typu *String*, vkládají další zprávy typu *String***.**

```
Date msgDate = message.messageDate;
String msgText = message.messageText;
String line = "(" + msgDate + ") " + msgText;
listModel.addElement(line);
```

Do této metody přijde atribut *message* typu *ChatMessage*, což je struktura obsahující datum poslání zprávy a text zprávy. V této metodě se vstupní data zformátují a následně vloží do proměnné *listModel*, která zajistí okamžité zobrazení zprávy v panelu.

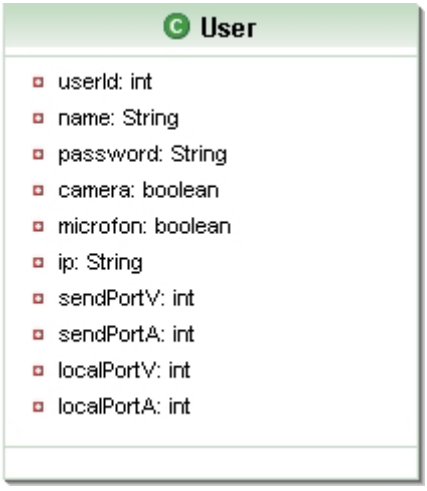
Metoda *displayHistory()* požádá server o vrácení seznamu všech poslaných zpráv a postupně pro tento seznam volá metodu *displayMessage()*.

```
public void displayHistory(SocketStub server) throws Exception {
     this.server = server;
     ChatMessage[] messages = server.getHistory();
     for (int i = 0; i < messages.length; i++) {
          displayMessage(messages[i]);
     }
}
```

7.2.2.4 UserPanel

Třída *UserPanel* se nestará pouze o vzhled panelu, ale nachází se v ní také veškerá logika související se správou uživatelů a portů. Správa uživatelů a portů spolu úzce souvisí. A to proto, že každý uživatel odvozený od instance třídy *User* zná porty, na kterých bude vysílat nebo přijímat.

Jakmile se klient připojí k serveru, vytvoří se instance třídy *User* a nastaví se její dané atributy, jako je unikátní identifikační číslo uživatele, jméno uživatele, IP adresa klienta, příznaky toho jestli má kameru a mikrofon, a porty na kterých pracuje. Z důvodu toho, že pro přenos nepoužívám *multicast*, ale *multi-unicast*, což znamená, že všem uživatelům posílám ta samá data, ale jen na různé IP a různé porty, každý uživatel pracuje na čtyřech portech. Aplikace samotná potřebuje jen dva, jeden pro video signál a druhý pro audio signál. Z principu fungování RTP protokolu, každý vysílaný signál, požívá ještě další kontrolní port.Více o vysílání a přijímání A/V signálu najdeme v kapitole 7.4.



Obr. 29: Atributy třídy User

Jakmile se kdokoliv přihlásí do aplikace, vytvoří se instance třídy *User*, a uloží se do seznamu uživatelů *userList***.** Tento seznam v sobě nese informace o všech uživatelích v konferenci. Seznam *userList* je důležitý pro správu portů, na kterých se bude vysílat a přijímat. Počáteční port, se kterým se začíná pracovat, zadává přednášející před začátkem vysílání. Od toho se odvíjí výpočet portů pro ostatní uživatele. Jako první se do seznamu přidá instance třídy *User* odpovídající přednášejícímu. To proto, že potřebujeme znát jeho atributy pro další operace, jako je jeho přijímání multimediálních dat od ostatních uživatelů, čili potřebujeme znát porty na kterých pracuje.

Jelikož každý uživatel zabere čtyři porty, tak pokud přednášející začíná na portu 5000, vysílá video signál na portu 5002 a audio signál na portu 5004. Porty 5001 a 5003 jsou už z principu protokolu RTP kontrolní porty. Další uživatel začíná pracovat na portu 5006.

Pro správu uživatelů jsou ve třídě *UserPanel* důležité tyto metody:

- **`public void addUserAndTargetToList(User user)`**

  Tato metoda, vloží do seznamu uživatelů *userList* uživatele, který do metody přijde jako parametr. Logika pro výpočet portu a logika získání informace a přihlášeném uživateli, se nachází ve třídě *SocketSkeleton*, v metodě *processLogon()*.

- **`public void connectWithSelectedUser(List<User> users2Connect, boolean mic, boolean cam)`**

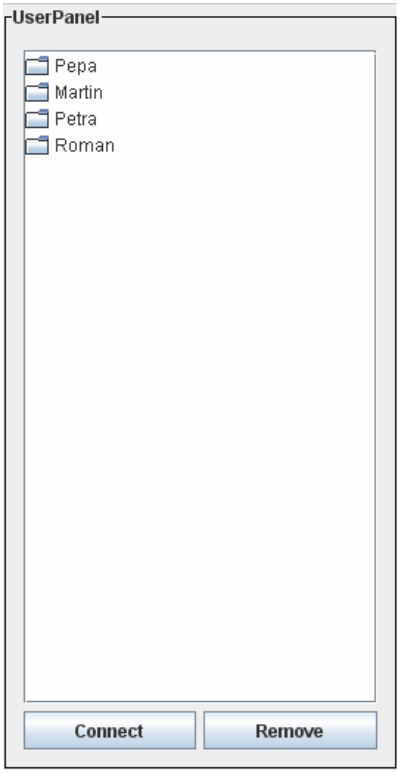
Tato metoda se zavolá pokud se chce přednášející individuálně spojit s daným uživatelem. Parametry této metody jsou seznam uživatelů, kterým začne daný uživatel vysílat, pak booleovská proměnná značící jakým způsobem se s klientem přednášející spojí.

- **`public void disconnectSelectedUser(User selectedUser)`**

Pokud se přednášející rozhodne pro ukončení individuálního spojení, zavolá se tato metoda. Jako parametr vstupuje do metody uživatel, kterému se pošle informace o ukončení vysílání.

- **`public synchronized void removeCurrentUserFromList()`**

Tato metoda se zavolá pokaždé, pokud nějaký uživatel odchází z konference. Buď se odpojí, nebo mu vypadne spojení , nebo se přednášející rozhodne vykázat z konference.



Obr. 30: Detail UserPanel

7.2.2.5 MenuPanel

MenuPanel je použit pro zobrazení menu. V menu jsou tři položky:

- **Preferences**

  Pomocí této položky může uživatel nastavit zdroje pro snímání videa a zvuku. Standartně jsou nastaveny tyto hodnoty:

  Video: **vfw://0 -** Zdroj je webkamera

  Audio: **javasound://0 -** Zdroj je mikrofon
- **Exit**

  Ukončí aplikaci
- **About**

  Zobrazí panel s informacemi o programu

7.3 Aplikace Klient

Aplikaci Klient spouští uživatel, který se chce vzdáleně připojit k aplikaci Server, komunikovat pomocí chatu s ostatními uživateli a hlavně přijímat multimediální data od přednášejícího uživatele. Veškerá logika je umístěna na serveru, klient pouze zobrazuje a zpracovává přijímaná data ze serveru.

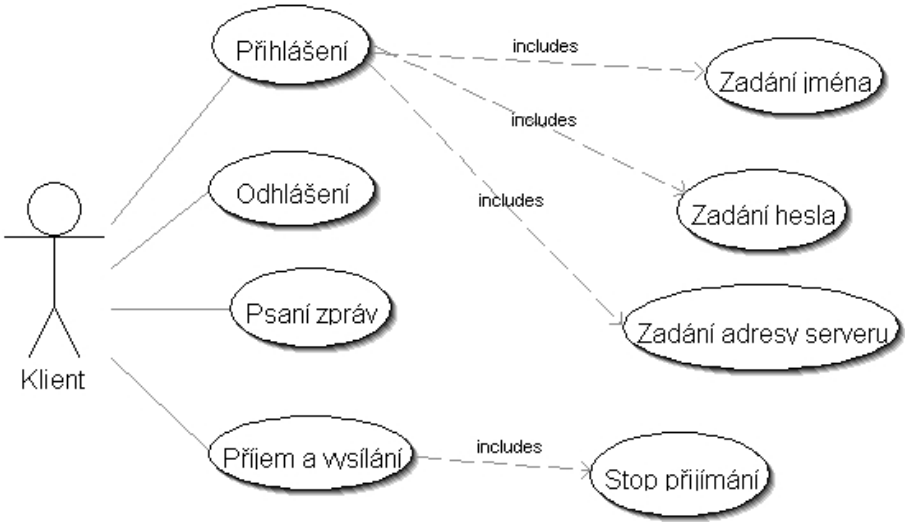
Hlavní funkce Klienta je příjem multimediálního vysílání ze Serveru. Pokud se chce přednášející spojit individuálně s daným klientem, klient po odsouhlasení požadavku začne vysílání stejně jako Server. Umožňuje také vysílání všem klientům zároveň pokud toto přednášející požaduje.

7.3.1 Seznámení s aplikací Klient

Uživatel aby se mohl přihlásit do konference, musí znát předem heslo. Toto heslo musí nějak zprostředkovat přednášející. Toto omezení je v aplikaci z důvodu bezpečnosti před nechtěnými uživateli.

Pokud uživatel zná heslo, může se přihlásit. Zadá své jméno, pod kterým bude v konferenci vystupovat, požadované heslo a IP adresu počítače na kterém běží aplikace server. Pokud je vše v pořádku, uživatel může s ostatními komunikovat pomocí chatu a přijímat A/V signál od přednášejícího.

Pokud má uživatel zajímavý dotaz nebo s ním chce přednášející něco podrobněji rozebrat, přednášející muže uživatele požádat o spojení. Pokud uživatel souhlasí a má k dispozici mikrofon nebo kameru začne vysílat svá multimediální data ostatním uživatelům.

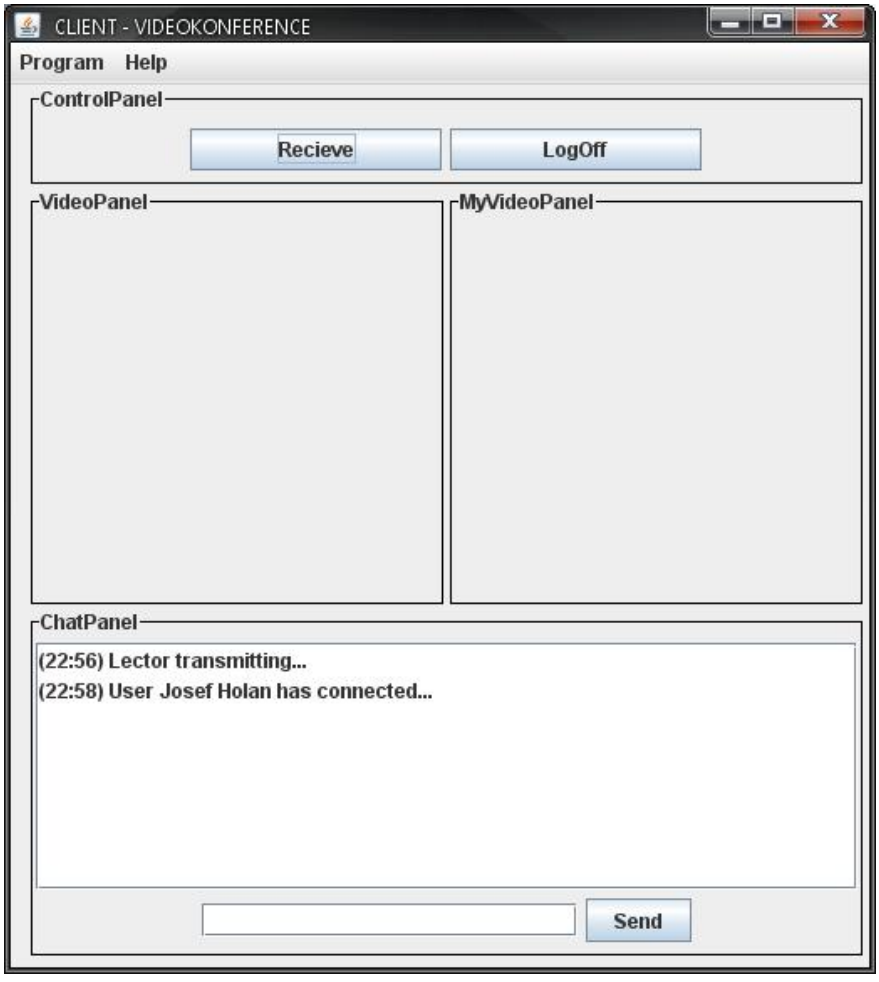


Obr. 31: Use Case Diagram aplikace Klient

7.3.2 Struktura

Aplikace Klient vychází, ze stejného návrhu jako aplikace Server. Základem celé aplikace je hlavní panel, do kterého se vkládají další komponenty jako je menu a další vnitřní panely. Tento panel je zkonstruován pomocí instance třídy *ClientGUI*.

Pro jednoduchost a přehlednost jsem celou aplikaci rozdělil do několika funkčních oblastí. Každou tuto oblast znázorňuje panel.



Obr. 32: Aplikace Klient

- **Control Panel**
  Panel s dvěmi tlačítky pro odhlášení z konference a pro zapínaní a přijímaní příjmu A/V signálu od Serveru.
- **Video Panel**
  Panel, na kterém uživatelé uvidí klienta, se kterým se přednášející rozhodl spojit.
- **MyVideo Panel**
  Panel, na kterém klient vidí a slyší vlastní obraz a zvuk z od přednášejícího.
- **ChatPanel**
  Panel pro písemnou komunikaci mezi uživateli. Do tohoto panelu se vypisují i informativní hlášky ze serveru.
- **MenuPanel**
  Panel pro zobrazení menu.

Na straně klienta jsou tyto třídy pojmenované stejně jako názvy panelů, slouží pouze pro grafické zobrazení panelu. Logika je implementována na straně aplikace Server. Data se přenáší pomocí distribuovaných objektů, popsaných podrobněji v kapitole 7.1

7.4 Přenos audia a videa

Ve videokonferenci pro přenos A/V signálu je použita knihovna Java Media Framework. Tato knihovna je podrobně popsána v kapitole 6. V mé aplikaci jsem pro přenos multimediálních dat použil defaultní protokol RTP. Tento protokol pracuje nad protokolem UDP.

V aplikaci funkčnost vysílače definuje třída *AVTransmiter* a funkčnost přijímače definuje třída *AVReciever*. Tyto dvě třídy obsahují veškerou funkčnost a metodiku pro práci s multimediálními daty.

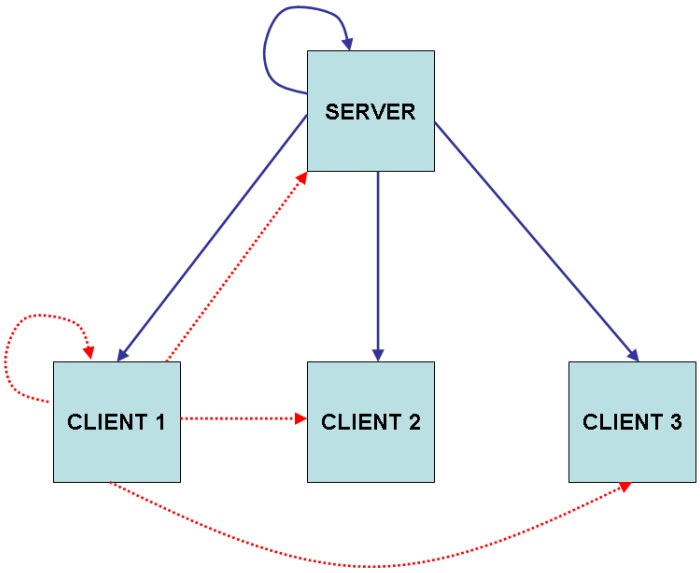
7.4.1 Logika vysílání a příjmu

Hlavní změnou oproti původnímu plánu, je použití vysílání tzv. *multi-unicastem* místo *multicastu*. K nepoužití *multicastu* jsem dospěl po mnoha problémech, se kterými jsem se musel potýkat. *Multi-unicast* jsem musel použít z důvodu nutnosti použití dvou procesorů pro vysílání audio a video signálu zvlášť. Tento signál je snímán z kamery a mikrofonu. Můj původní návrh byl, tyto dva oddělené streamy spojit do jednoho a vysílat jako jeden stream. Po naimplementování této logiky jsem zjistil, že tento způsob naplno vytěžuje procesor a pro aplikaci je tento způsob nepoužitelný.

Proto jsem přenos audia a videa oddělil od sebe a to tak, že každý stream má svůj procesor. Každý procesor zpracovává svůj stream a vysílá jej odděleně na určitém portě. Tyto dva procesory a streamy, ale musíme synchronizovat tak, aby si časově odpovídaly tak, aby si každý stream s časovou závislostí odpovídal.

Při startu vysílání, začne nejprve server vysílat sám sobě. To proto, aby přednášející mohl sledovat obraz sebe samého. Nejdříve jsem tuto logiku implementoval tak, že jsem používal klonované *DataSource* (viz. Kapitola 3). Tato možnost, ale byla nepoužitelná z důvodu velkého vytížení procesoru a zpomalení celé aplikace.

Při každém přihlášení nového klienta, se dodá serveru informace o daném klientovi jako je IP adresa, a port, na který se mu bude vysílat. Daný port se mění, a proto se pošle informace pro příjem daného streamu také klientovi. Jakmile server zná dané hodnoty, začne vysílat na daný port a IP adresu. Klient na tom samém portu začne přijímat. Stejně pracuje i vysílání od klienta, se kterým se rozhodl přednášející spojit. Graficky je to znázorněno na obrázku číslo 33.



Obr. 33: Princip vysílání

7.4.2 AVTransmitter

Třída *AVTransmitter* se stará vysílání A/V signálu. Hlavní metoda, která inicializuje a zároveň startuje přenos je metoda *start()*. Tato metoda nejdříve vytvoří a zinicializuje oba procesory zpracovávající jednotlivé streamy, video a audio stream. Poté vytvoří vysílač metodou *createTransmitter()*. Tato metoda zinicializuje *RTPManager*, který se stará o chod vysílání.

```
public synchronized String start(...) {
            // ...
            result = createVideoProcessor(videoFileName);
            // ...
            result = createAudioProcessor(audioFileName);
            // ...
            result = createTransmitter(user);
            // ...
            // Start the transmission
            videoProcessor.start();
            audioProcessor.start();

            return null;
            }
```

Metoda *createTransmitter()* pro každý procesor vytvoří *RTPManagera*, který se stará o chod každého procesoru. Nejdříve se vytvoří instance daného manageru, poté se neinicializuje a tím se mu zadá IP adresa a port na které začne vysílat.

```
private String createTransmitter(...) {

      // ...
      rtpMgrs[0] = RTPManager.newInstance();

      rtpMgrs[0].initialize(new SessionAddress(InetAddress, port));

      rtpMgrs[0].addReceiveStreamListener(this);
      rtpMgrs[0].addRemoteListener(this);


      this.addTarget(port, rtpMgrs[0], InetAddress, sendPort);
      sendStream = rtpMgrs[0].createSendStream(videoDataOutput, 0);
      sendStream.start();


      // ...

      return null;
}
```

Manager pracuje tak, ze jakmile začne vysílat, můžeme mu kdykoliv přidat další cíl, na který začne vysílat. Další cíl se do vysílače přidává metodou *addTarget()*. Po přidání dalšího cíle vysílač začne vysílat automaticky na daný cíl.

```
SessionAddress addr = new SessionAddress(InetAddress, port);
mgr.addTarget(addr);
```

7.4.3 AVReciever

Třída *AVReciever* se stará o příjem vysílaného signálu. Základem logiky pro příjem, je vytvoření instance *RTPManager*. Ten se zinicializuje nastavením IP adresy portu, se kterými bude pracovat, čili přijímat. Tato logika se nachází v metodě *addTarget()*. Zároveň se vytvoří instance typu *Player*, která zobrazuje přijímaný A/V signál.

```
public void addTarget( ... ) {

    // ...

    RTPManager mgr = RTPManager.newInstance();

    //...

    ipAddr = InetAddress.getByName(senderAddress);

    int local_port = new SessionAddress(InetAddress, port);
    int remotePort = new Integer(senderPort).intValue();
    int destAddr = new SessionAddress(ipAddr, remotePort);
    }

    mgr.initialize(localAddr);

    BufferControl bc = mgr.getControl("BufferControl");
    if (bc != null) {
          bc.setBufferLength(35);
    }

    mgr.addTarget(destAddr);

    mgrs.add(mgr);

    // ...
}
```

Pokud chceme, aby příjemce přestal přijímat data z určitého vysílače, zavolá metodu *removeTarget()*. Ta nejdříve zavře okno, ve kterém se zobrazuje daný signál a poté vyhledá daný *RTPManager* a odebere mu ze seznamu, adresu a port na kterém daný signál přijímá.

```
public void removeTarget(String address, String port) {
    for (int i = 0; i < playerWindows.size(); i++) {
         PlayerWindow pw = (PlayerWindow) playerWindows.get(i);
      if (address.equals(pw.senderAddress) && port.equals(pw.senderPort)) {
            // ...
            pw.close();
            playerWindows.remove(pw);
            break;
      }
   }
   for (int i = 0; i < mgrs.size(); i++) {
      // ...
      if (address().equals(address) && port.equals(dataPort)) {
            mgr.removeTarget(addr);
            mgr.dispose();
            mgrs.remove(mgr);
            break;
      }
   }
}
```

8 Závěr

Cílem této diplomové práce bylo vytvořit a popsat videokonferenční systém, který by měl být využitelný jako prostředek e-learningu. Pro implementaci jsem použil knihovnu Java Media Framework, která umožňuje práci s multimediálními daty přenášenými v reálném čase. Její obecný popis lze nalézt v kapitole číslo 3. Dále jsem se zajímal o problematiku protokolů RTP/RTCP, které jsou nezbytné pro přenos multimediálních dat v reálném čase. Data, která jsou posílána v dotazu od studenta jsou přenášena sítí pomocí protokolu TCP. Proto je také jedna kapitola věnována právě tomuto protokolu.

Program jsem naimplementoval tak, aby byl přenositelný mezi různými operačními systémy a dal se spustit na jakémkoliv počítači připojeném do sítě LAN nebo internetu. Pro přenos multimediálních dat se využívá protokol RTP a s ním související protokol RTCP.

Při implementaci ukázkového programu jsem se potýkal s mnoha problémy. Poslední verze Java Media Frameworku vyšla v roce 2003 a vyskytuje se v ní spousta chyb. Možná pár chyb způsobovalo použití Javy 1.6 a její zpětná nekompatibilita s knihovnou JMF kompilovaná na Javě verze 1.4. Zjistil jsem, že knihovna JMF je velmi nestabilní a velmi hardwarově náročná i při použití současného výkonného hardwaru. Spoustu vysílací logiky jsem musel implementovat jinak než jsem zamýšlel. Nejdůležitější je nepoužití multicastu, ale multi-unicastu. Ten jsem musel použít kvůli velké hardwarové náročnosti, když jsem potřeboval spojit dva streamy z webkamery a mikrofonu. Přenos obou kanálů jsem uskutečnil pomocí použití dvou nezávislých procesorů (viz kapitola 7.4) a nezávislého zpracováni a přenosu obou streamů. Tyto streamy jsem poté časově synchronizoval, aby se jeden kanál nepředbíhal před druhým. Použití těchto dvou vysílacích procesorů mi znemožnilo použití multicastu kvůli použití více portů a správě obou procesorů.

Technologie Java Media Framework byla pro tento systém předpokládána již v zadání diplomové práce. JMF však pro implementaci náročnější aplikace nedoporučuji používat, hodí se totiž hlavně pro implementaci jednoduchého vysílání a příjmu multimediálních dat, jako například videa uvnitř Java appletu na internetové stránce.

Svou práci nedoporučují dále rozšiřovat a rozvíjet z důvodu velké nestability a zastavené podpory JMF ze strany firmy Sun. Z toho plyne také velká chybovost a hardwarová náročnost aplikace, kterou zapříčiňují především chyby a náročnost Java Media Frameworku. Pokud bych implementoval videokonferenční systém znovu, použil bych jazyk C++ nebo C#. Ne kvůli jazyku Java, ale kvůli absenci další konkurenční knihovny pro implementaci přenosu multimediálních dat.

9 Literatura

[1] Oficiální internetové stránky Javy, http://java.sun.com

[2] Pištěk, P. Multicast: skupinové vysílání. Zpravodaj ÚVT MU. ISSN 1212-0901, 1998, roč.8, č.5, s.13-15

[3] Dostálek, L., Kabelová, A.: Velký průvodce protokoly TCP/IP a systémem DNS. Computer Press, 2002. ISBN 80-7226-676-6.

[4] Pužmanová, R., Streaming media, DSL [online], 2004 http://www.dsl.cz/index.php?akce=188

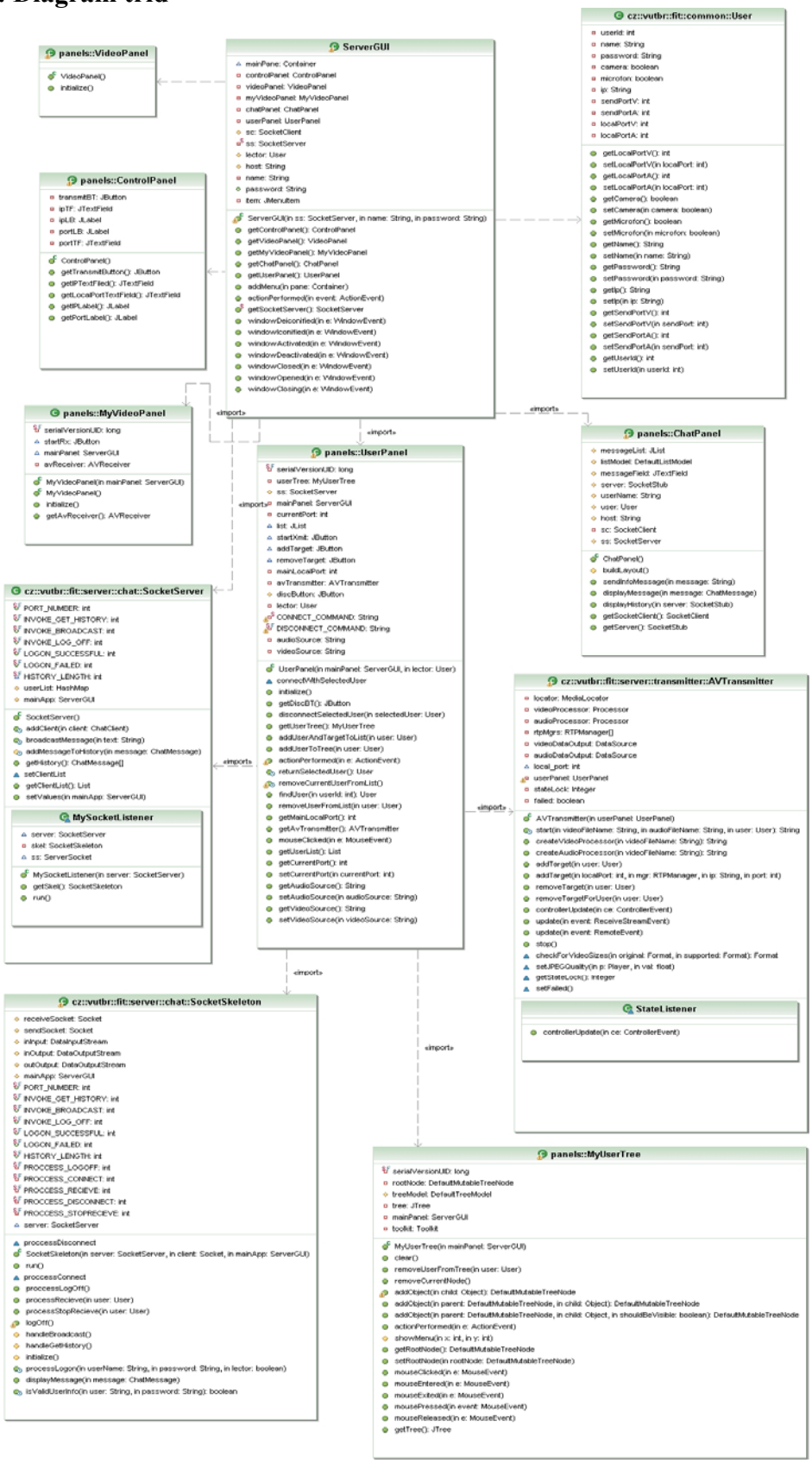
[5] Java Media Framework quide – http://java.sun.com/products/java-media/jamf/index.html

[6] Dan Komosný, Ph. D., Nové smery vývoje protokolu RTP/RTCP pro rozsáhlé konference v Internetu, http://www.elektrorevue.cz/clanky/04052/

[7] Wikipedia, www.wikipedia.org

10 Přílohy

Příloha 1: Diagram tříd



Příloha 2: Uživatelská příručka

Uživatelská příručka je přiložena na CD

Příloha 3: CD s vytvořenou aplikací

Důležité informace jsou v souboru README.TXT