Streaming Transport

Author: Susanne Burtzlaff, Schleißheimer GmbH

Revision: 0.0

Status: draft

File: CONCEPT\_StreamingTransport.docx

# History

|  |  |  |  |
| --- | --- | --- | --- |
| **Revision** | **Date** | **Author, Editor** | **Reason** |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# Table of Contents

[1 History 2](#_Toc458187121)

[2 Table of Contents 3](#_Toc458187122)

[3 Scope 5](#_Toc458187123)

[4 Theoretical Background 6](#_Toc458187124)

[4.1 Unicast vs. Multicast 6](#_Toc458187125)

[4.2 Muxing (Multiplexing) vs. MultiStreaming 7](#_Toc458187126)

[4.3 Internet Protocol Suite 8](#_Toc458187127)

[4.4 Transport Layer 9](#_Toc458187128)

[4.4.1 TCP: Transmission Control Protocol 9](#_Toc458187129)

[4.4.2 UDP: User Datagram Protocol 9](#_Toc458187130)

[4.5 Application Layer 9](#_Toc458187131)

[4.5.1 RTP: Real Time Transport Protocol 9](#_Toc458187132)

[4.5.2 RTCP: Real Time Control Protocol 10](#_Toc458187133)

[4.5.3 RTSP: Real Time Streaming Protocol 10](#_Toc458187134)

[4.6 Conclusion 10](#_Toc458187135)

[5 Concept 11](#_Toc458187136)

[5.1 Deployment 11](#_Toc458187137)

[5.2 Components 12](#_Toc458187138)

[5.3 Collaboration 13](#_Toc458187139)

[5.4 Messaging Sequences 14](#_Toc458187140)

[6 RTP libraries – state of the art 15](#_Toc458187141)

[7 Realization 16](#_Toc458187142)

[7.1 Streaming Module 16](#_Toc458187143)

[7.1.1 Initialise 16](#_Toc458187144)

[7.1.2 DoRender 19](#_Toc458187145)

[7.1.3 DeInitialise 20](#_Toc458187146)

[7.2 Receive Module 21](#_Toc458187147)

[7.2.1 Initialise 21](#_Toc458187148)

[7.2.2 DoRender 22](#_Toc458187149)

[7.2.3 DeInitialise 23](#_Toc458187150)

[7.3 Connector Module 24](#_Toc458187151)

[8 Outlook – next steps 25](#_Toc458187152)

[9 Appendix 26](#_Toc458187153)

[9.1 Details about Application Layer 26](#_Toc458187154)

[9.1.1 Definitions 26](#_Toc458187155)

[9.1.2 RTCP 27](#_Toc458187156)

[9.1.3 RTSP 27](#_Toc458187157)

# Scope

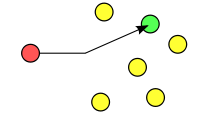
With the upcoming PF3NG project SSH is facing new challenges. Not alone the transfer of the responsibilities, also the infrastructure for a good cooperation between SSK, BSK, S&SE and Conti has to be established. Also a better integration of EmSo into the SSH workflow is a pressing issue.

One part to settle these challenges is to establish a pool of remote servers for operating the hardware at SSH. To allow several team members at different locations to work on a project it is mandatory to give all of them access to the respective hardware they are working on at the same time. The idea is to have one of the clients operating the system via a remote connection and broadcasting of a live video stream of that at a low latency and a best possible quality to his team members.

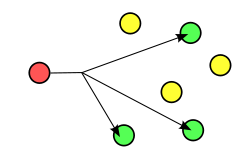
# Theoretical Background

## Unicast vs. Multicast

Unicast is a direct 1 on 1 connection between client and server. Each unicast connection is adding additionally to the complete data load.

IP multicast is a technique for one-to-many and many-to-many real-time communication over an IP infrastructure in a network. It scales to a larger receiver population by requiring neither prior knowledge of a receiver's identity nor prior knowledge of the number of receivers. Multicast uses network infrastructure efficiently by requiring the source to send a packet only once, even if it needs to be delivered to a large number of receivers. The nodes in the network (typically network switches and routers) take care of replicating the packet to reach multiple receivers such that messages are sent over each link of the network only once. There is no direct connection between clients and servers. This can save some bandwidth.  
An IP multicast group address is used by sources and the receivers to send and receive multicast messages. Sources use the group address as the IP destination address in their data packets. Receivers use this group address to inform the network that they are interested in receiving packets sent to that group.

: Unicast schematic   
[https://commons.wikimedia.org/wiki/File:Unicast.svg#/media/File:Unicast.svg]

* Unicast:  
  The media is transmitted to the source of the RTSP request, with the port number chosen by the client. Alternatively, the media is transmitted on the same reliable stream as RTSP.
* Multicast, server chooses address:  
  The media server picks the multicast address and port. This is the typical case for a live or near-media-on-demand transmission.
* Multicast, client chooses address:  
  If the server is to participate in an existing multicast conference, the multicast address, port and encryption key are given by the conference description.
* : Multicast schematic  
  [https://commons.wikimedia.org/wiki/File:Multicast.svg#/media/File:Multicast.svg]

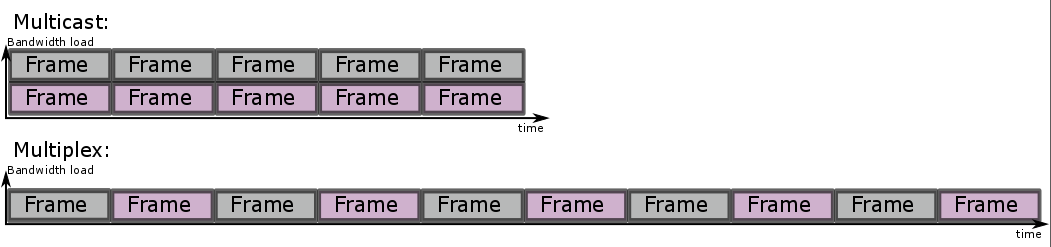
In unicast routing, each router examines the destination address of an incoming packet and looks up the destination in a table to determine which interface to use in order for that packet to get closer to its destination. The source address is irrelevant to the router. However, in multicast routing, the source address (which is a simple unicast address) is used to determine data stream direction. The source of the multicast traffic is considered upstream. The router determines which downstream interfaces are destinations for this multicast group (the destination address), and sends the packet out through the appropriate interfaces. The term reverse path forwarding is used to describe this concept of routing packets away from the source, rather than towards the destination.

For obvious reason for SSH Streaming Transport will use Multicast connection with allowing the server to choose the addresses.

## Muxing (Multiplexing) vs. MultiStreaming

Multiplexing interleaves several signals into one single data stream.

The resulting transport stream (MPEG-TS, MTS or TS) is a standard container format for transmission for transmission and storage of audio, video, and Program and System Information Protocol (PSIP) data.



URL1 🡪

URL2 🡪

URL1 🡪

: Multistreaming vs. Multiplexing (Muxing)

Muxing is useful for low end clients with slow throughput (such as over a slow network connection). Multiplexing is the process of alternately sending (interleaving) data.

Multistreaming is on the other end of the spectrum where there is a high-end client. Multistreaming allows multiple streams of a single client to occur simultaneously to multiple tape drives.

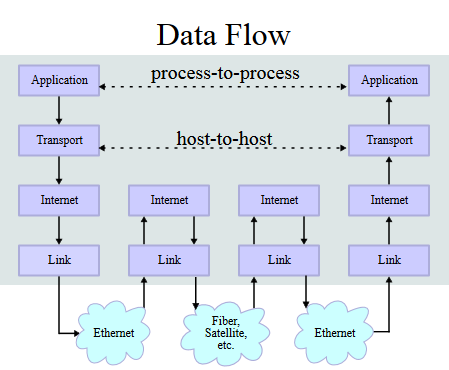
**A combination of multiplexing and multistreaming is possible.** The right combination can reduce transmission times of large data loads. Be aware that multiplexing will reduce the performance of the streams. The larger the multiplexing factor, the greater the impact on restore performance.

* Muxing is slowing down the transmission speed. We need to perform some tracings to know if we might need it or not and if so, up to which degree.

## Internet Protocol Suite

The Internet protocol suite is the computer networking model and set of communication protocols used on the internet and similar computer networks.

The functionality of the internet protocol suite is encapsulated into divisions of layers of general functionality. The Data Flow is represented below.



: Data Flow concept [By en:User:Kbrose - Prior Wikipedia artwork by en:User:Cburnett, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=1831900]

Since we are not about to make an internet application we can focus on the application and transport layers for the SSH Streaming Transport Concept.

The following descriptions of protocols, which are to be used for this module, refer to the Request For Comments (RFC) which are provided by The Internet Engineering Task Force (IETF®).

## Transport Layer

### TCP: Transmission Control Protocol

Ref: RFC 793 (<https://tools.ietf.org/html/rfc793>)

Responsibility: Providing reliable, ordered, and error-checked delivery of a stream of octets between applications running on hosts communicating over an IP network.

TCP has an additive 16-bit checksum and it favors reliability over timeliness. It can handle only a single stream of data per connection. This is enough to transmit some initial information as well as for controlling state of the data streaming. But we want to make sure we do not lose a packet in the process.

### UDP: User Datagram Protocol

Ref: RFC 768 (<https://tools.ietf.org/html/rfc768>)

Responsibiltiy: Providing simple connectionless transmissions with a minimum of protocol mechanism. Due to not having handshaking dialogues there is no guarantee of delivery, ordering or duplicate protection. UDP provides checksums for data integrity and port numbers for addressing different functions at the source and destination of the datagram.

This is the right protocol to use to transport the frame wise video and audio bit streams. It is important that the transfer of many data is being processed fast. If we might lose a packet in between this is beyond human recognition and should not matter much. We will use application based concealment algorithms to cover that. In a later stage of the Remote Workstation Concept a Recorder on the server side is scheduled to enable data analysis after the recording.

## Application Layer

### RTP: Real Time Transport Protocol

Ref: RFC 3550 (<https://tools.ietf.org/html/rfc3550>)

Responsibility: Delivering audio and video via IP networks. Real-time, end-to-end transfer of streaming media.

The protocol provides facilities for jitter compensation and detection of out of sequence arrival in data, which are common during transmissions on an IP network.

Real-time multimedia streaming application require timely delivery of information and often can tolerate some packet loss to achieve this goal (by using error concealment algorithm). The TCP, although standardized for RTP use, is not normally used in RTP applications because. Instead the majority of the RTP implementations are built on the UDP.

Note that **RTP itself does not provide any mechanism to ensure timely delivery** or provide other quality-of-service guarantees, but relies on lower-layer services to do so. It does not guarantee delivery or prevent out-of-order delivery, nor does it assume that the underlying network is reliable and delivers packets in sequence. The sequence numbers included in RTP allow the receiver to reconstruct the sender's packet sequence, but sequence numbers might also be used to determine the proper location of a packet, for example in video decoding, without necessarily decoding packets in sequence. (Ref. RFC 3550, 1.)

### RTCP: Real Time Control Protocol

Ref: RFC 3550, RFC 3605 (<https://tools.ietf.org/html/rfc3550>, <https://tools.ietf.org/html/rfc3605>)

Responsibility: Providing out-of-band statistics and control information for an RTP session.

The primary function of RTCP is to provide feedback on the quality of service (QoS) in media distribution by periodically sending statistics information to participants in a streaming media session.

If **both audio and video media** are used in a conference, they are **transmitted as separate RTP sessions**. That is, separate RTP and RTCP packets are transmitted for each medium using two different UDP port pairs and/or multicast addresses. There is no direct coupling at the RTP level between the audio and video sessions, except that a user participating in both sessions should use the same distinguished (canonical) name in the RTCP packets for both so that the sessions can be associated.   
Synchronized playback of a source's audio and video can be achieved using timing information carried in the RTCP packets for both sessions. (Ref. RFC 3550, 2.2)

### RTSP: Real Time Streaming Protocol

Ref: RFC 2326 (<https://tools.ietf.org/html/rfc2326>)

Responsibility: Establishing and controlling media sessions between end points.

Clients of media servers issue VCR-style commands (e.g. play, record, pause) to facilitate real-time control of the media streaming from server to a client or from a client to a server.

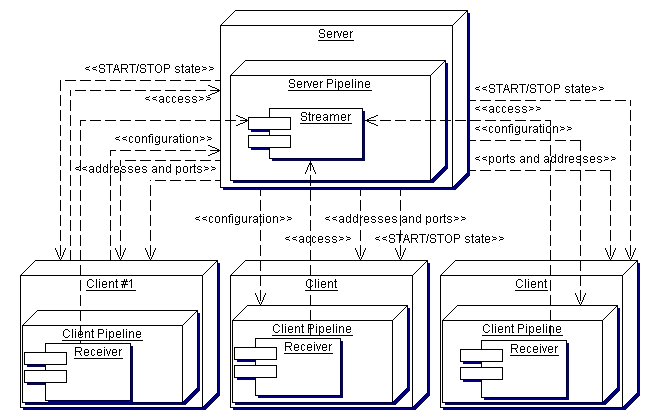
## Conclusion

We got a good overview now how the different protocols work and interact. Some further information are available in the appendix. Check out chapter 9.1.1 or 9.1.3.1 if you have trouble with a technical term used in this document.

# Concept

## Deployment

The streaming transport concept is focusing on establishing the connections and transferring the data from the server to several clients. The main operator client (henceforth called Client #1) is sending configurations to the server. According to that the pipeline will be configured and the necessary addresses and ports will be chosen. The server is starting its pipeline then and is broadcasting the video and audio data to those addresses and ports. Any client can now send a request to the server to receive the addresses and ports to establish connection and the configuration to configure its own pipeline accordingly. Once the connection is established the client can access the streamed data.



: Server streaming to several clients

## Components

Before any streaming connection is going to start the client is connecting to the server via TCP on a fixed port on the server’s IP by pushing the START button in the Control GUI.

The Connector Client #1 is sending the configuration which the server is supposed to use for the pipeline. The Connector Server is sending back the addresses and ports which will be used to stream the media (video and maybe audio) it is creating. Per default the Connector Server is also going to send back the configuration. In case another client is joining the multimedia session later on it will receive the configuration for its pipeline to be able to process the stream appropriately.

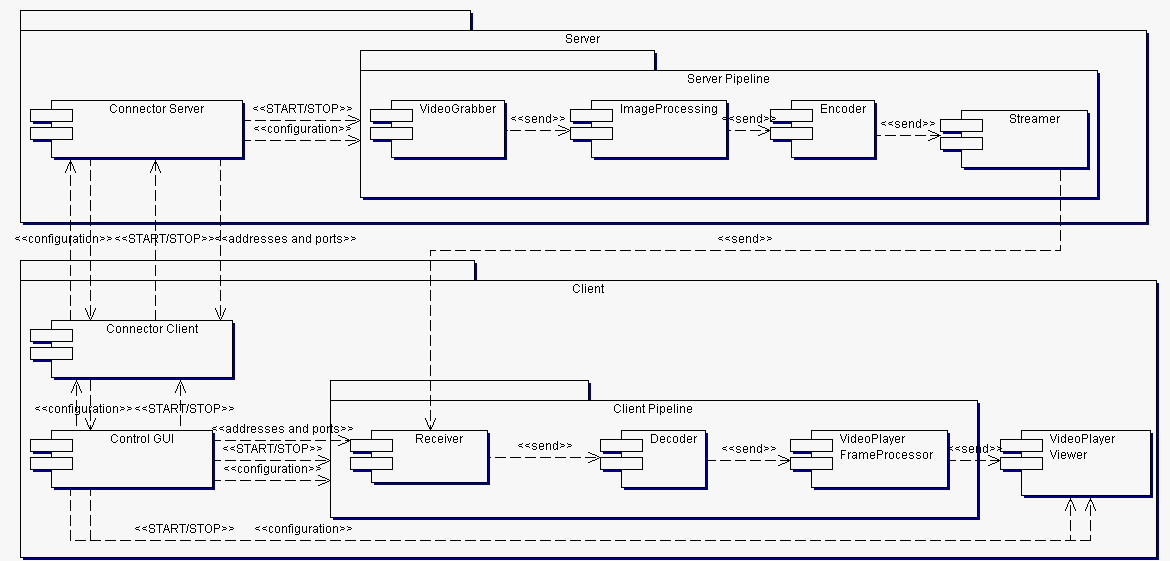
By pushing the START button the Client #1 also enabled his own pipeline and VideoPlayer Viewer as well as the pipeline of the server to start. Every other client has only the possibility to control his own pipeline and VideoPlayer. The same applies for the STOP button.

The Server Pipeline will be created according the received configuration. The modules VideoGrabber, ImageProcessing and Encoder are already described in the Concept Remote-Workstation.

The Streamer is creating streams out of the data it received from the prior modules. It broadcasts these streams to the addresses and ports which have been communicated before to the clients.

The clients have started their pipeline according the configuration they received via the Connector. They open the ports for the stated addresses and receive the streams. They pass the data they extracted out of the streams to the modules to follow.

The pipeline will pass video and audio data to the VideoPlayer Viewer for the client user to finally observe the multimedia session.



: Components of server and client

## Collaboration

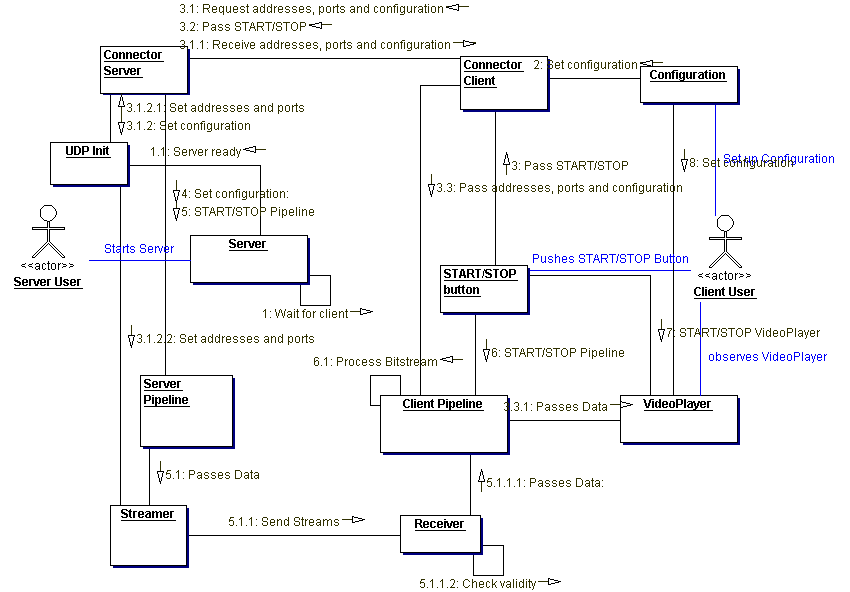
The Client is setting the configuration (if he is Client #1), pushing the START/STOP button and is observing the VideoPlayer.

If the configuration is set and the START button is pressed the client establishes a connection to the server (IP address of the server, fixed ports for this request).

In case the client is Client #1 the configuration will be send to the connector of the server, causing the initialization of the UDP connection. The addresses and ports are created and will be provided for every client to follow as well as the configuration. If any other client is establishing now a connection those addresses, ports, the configuration as well as the START/STOP status of the server will be transmitted.

With the received configuration the server is starting its pipeline. The created data are passed to the streamer which will use the UDP connection to send them to the client receiver.

The START button push caused the client to start its own pipeline with the configuration the connector passed on. The pipeline receives the data of the server and processes the bitstream to create video and maybe audio data again which will be displayed in the VideoPlayer.

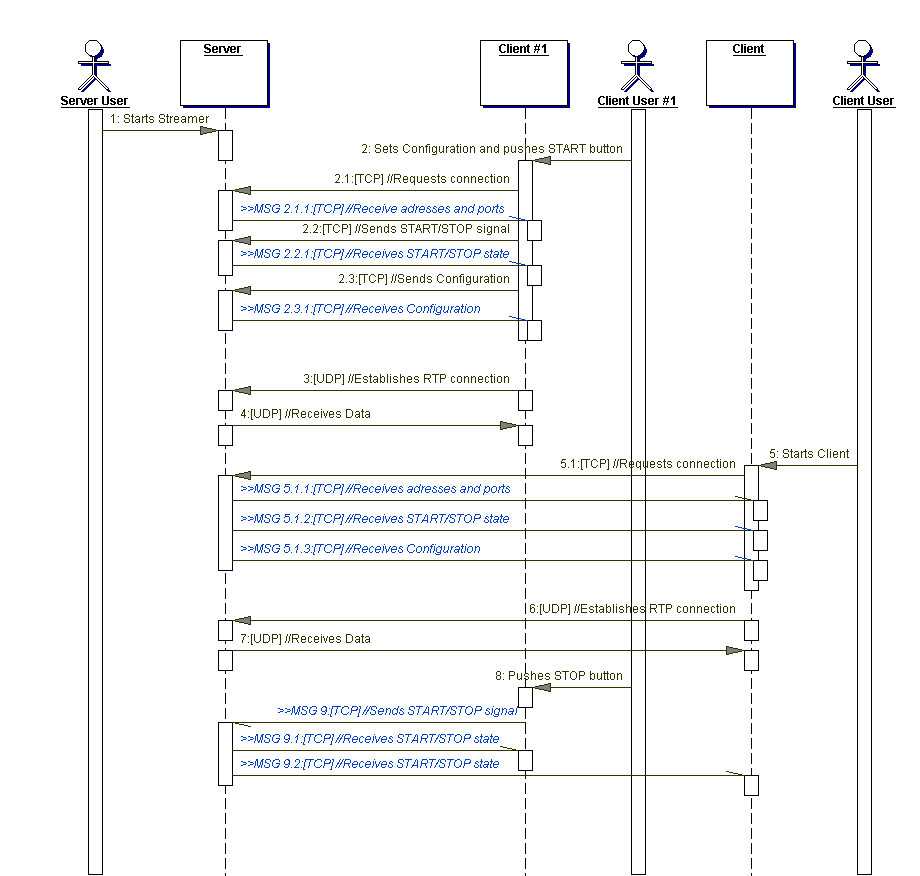


: Collaboration of Streaming Transport components

## Messaging Sequences

So the only way to communicate between server and clients is the TCP connection for the information transfer and the UDP connection for the data transfer. Client #1 is setting the configuration for the multimedia session and sets the running state of it. The initial TCP connection is setting up the multimedia session and preparing the UDP connection by creating addresses and ports for every RTP and RTCP session.

The UDP connection can be established now and the server is sending data to every client as long as no STOP signal is being transmitted asynchronously to the pipelines.



: Messaging Sequences

# RTP libraries – state of the art

We want the library we are going to use to handle a lot of that protocol processes internally. Let’s evaluate the available libraries!

Some fields which might pose some possible issues in the later usage of the library are marked.

: Overview of Streaming supporing libraries

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Library** | **Developers/Community** | **Language/Plattform** | **hevc?** | **License** | **other** |
| JRTPLIB | Jori Liesenborgs  (made this lib during his PhD thesis)  Last version November 2011 | C++  CMake | yes | MIT. Completely free without any warranty. “AS IS”. | Best to use together with JTHREAD  Support for the Real-time Transport Protocol (RTP) |
| PJSIP | [Teluu Ltd.](http://www.teluu.com)  (small business in UK) | C  NMake | no | GNU General Public License | Support for SIP, SDP, RTP, STUN, TURN, and ICE. Multimedia framework. |
| MediaSuite.NET | StreamcodersTM | C#/.NET | yes | MediaSuite has a flat price per developer machine | Multimedia Framework for Microsoft .NET (Streaming protocols, video codecs, encapsuilation, image processing, image filters, audio, networking, …) |
| Live555 Streaming Media | [Ross Finlayson](http://www.rossfinlayson.com/)  Since 2006.  Last version May 2016 | C++  NMake | yes | GNU Lesser General Public License | Unicast and multicast streaming, Streaming Server and Client, UDP or TCP. Basic functionalities we need. |
| libav | Libav Team  (free OpenSource developers)  Split up from FFmpeg developer community in March 2011 | C  Make  (a pain to build on Windows) | yes | GNU Lesser General Public License v2.1 | Extensive: Provides cross-platform tools and libraries to convert, manipulate and stream a wide range of multimedia formats and protocols |
| FFmpeg | [Fabrice Bellard](https://en.wikipedia.org/wiki/Fabrice_Bellard) and FFmpeg Team  (free OpenSource developers) | C  Make  (a pain to build on Windows) | yes | GNU Lesser General Public License v2.1 | Extensive: Opensource multimedia framework. Streaming server, player, libraries for audio and video compression, general purpose image and video processing libraries. |

The JRTPLIB library seems not to be under development anymore. Additionally it seems it was not used by many projects. The PJSIP libraries do not provide HEVC (H.265) support which is scheduled in the Remote Workstations Concept. The MediaSuite.NET libraries is written and aimed for usage in C#/.NET. It is possible to get around that by including prebuilt libraries but it is probably going to be nasty to do. FFmpeg and libav are much too extensive for our purpose. They are focusing mostly on codecs and pipeline handling.

We do not need a server which is broadcasting any available video on the hard drive. We need a streamer which is streaming frame wise input and which is adjustable to our needs. And a respective receiver of course too …

That is what Live555 Streaming Media is providing. All we need.

# Realization

## Streaming Module

For SSH Streaming Transport Concept we will establish a module for a multistreaming RTSP consisting of a media session that may include several RTP sessions as a first step. We will have the option to either stream several RTP sessions in a media session or to stream one muxed stream in a media session.   
It is possible to use several media sessions for one RTSP server, in principle.

The streaming concept for multistreaming and for muxed streaming is presented here. The main difference is that in multistreaming each RTP session has each a different URL and is streamed parallel. For muxed streaming an array of track states is added to one RTP session with one URL.

Each Multimedia Session consists of one or more RTP sessions.

We use for multistreaming sessions some code snippets from the example testH264VideoStreamer.

**Muxed streaming:** We need an array of structures representing the state of the video, audio and/or subtitle tracks. Something like e.g.

static struct {

unsigned trackNumber;

FramedSource\* source;

RTPSink\* sink;

RTCPInstance\* rtcp;

} trackState[n];

**You have to loop through every trackState as if it were a RTP session on one destination adress only but several ports.** MultiStreaming sessions may contain several RTP session with a unique destination adress each.

### Initialise

At the beginning of every MediaSession the BasicTaskScheduler and the basicUsageEnvironment has to be set. We have to choose a random IPv4 SSM address. A ServerMediaSession has to be created.

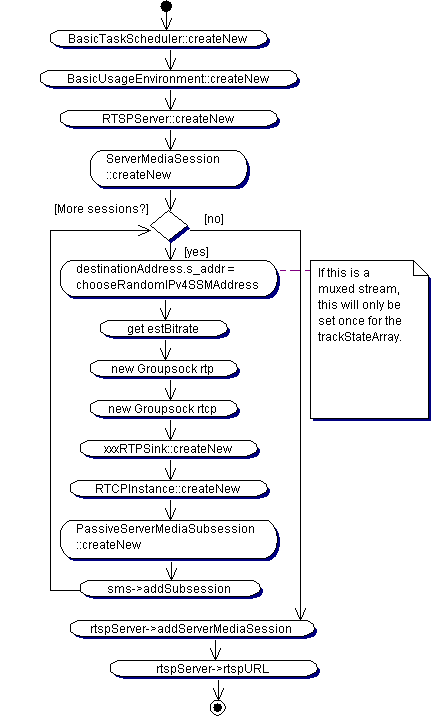
**Muxed streaming:** For muxed streaming only one destination address for the whole trackState array will be used. We use createSourceForStreaming for returning the proper estBitrate (in testMKVStreamer is an example). For each media type, the estBitrate is fixed. createSourceForStreaming is also creating a new VideoStreamDiscreteFramer if the track is an h264 or h265 video.

We create a new destination address for every new RTP session with chooseRandomIPv4SSMAddress. We need to create RTP and RTCP groupsocks to create a new sink for the video or audio type in question and to create a new RTCPInstance. We set the RTPSource as NULL, since we are server. Creating the RTCPInstance will start RTCP running automatically.

**Muxed streaming:** The rtpPortNumber and the rtcpPortNumber for creating the groupsocks need to have different values for every item in the trackState. We create the sink with createRTPSinkForTrackNumber. Since we cannot read the trackNumber out of an existing file, we will need to reimplement the functions createSourceForStreaming and createRTPSinkForTrackNumber.

With the sink and the RTCP instance, we are creating a PassiveServerMediaSubsession that we are adding then as subsession to the ServerMediaSession.

The complete ServerMediaSession itself will be added to the RTSP server and announced using rtspURL, which will return the URL of the stream.



: Initialise of the server multimedia session and the several RTP sessions

### DoRender

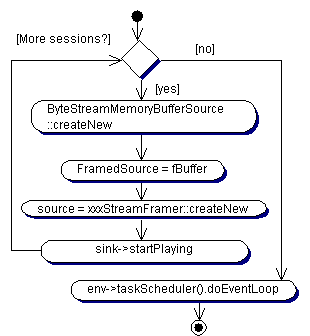
Since we are using sequential added buffer we have to use ByteStreamMemoryBufferSource. The bitstream buffer from the previous module is an input parameter for createNew.

Set the buffer as ByteStreamMemoryBufferSource as FramedSource and create a StreamFramer according the video or audio type.

startPlaying will read out buffer until it is empty. It delivers the function afterPlaying. The function doEventLoop is a repeatedly looping and handling readable sockes and timed events. The live555 libraries are working asynchronously and hence the streaming process restarts for every frame.

void afterPlaying(void\*) {

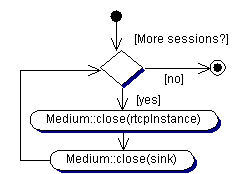
sink->stopPlaying();  
 Medium::close(source);  
}



: DoRender for the server RTP sessions

### DeInitialise

We call stopPlaying in DeInitialise. Meaning, we are telling the module when the data stream will end for each sink. Closing the rtcpInstance will send the BYE signal.



: DeInitialise for the server RTP sessions

## Receive Module

We are going to use the same libraries for realizing the client as we do for the streamer for obvious reasons. Therefore, chapters 4 and 6 apply here the same way.

We are using code snippets from the examples testMPEG1or2VideoReceiver, testMPEG2TransportReceiver or testMP3Receiver.

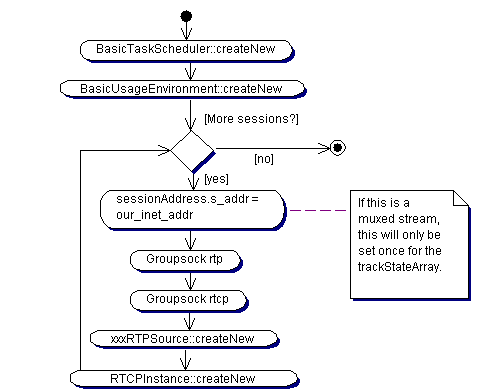
**To establish a connection the client needs prior knowledge about the session addresses and ports of the server.**

For the client we need a structure to hold the state of the current session. In case of a multistreaming session we can use an array of sessions to receive data. startPlaying is delivering this function in DoRender.

### Initialise

We create a new BasicTaskScheduler and BasicUsageEnvironment. We create RTP and RTCP groupsocks with the address and ports from the server, to create an RTPSource for the video or audio type in question and to create a new RTCPInstance. As RTPSink we set NULL since we are client. Creating the RTCPInstance will start RTCP running automatically.

**Muxed streaming:** The session address will be the same for the whole trackState array. Only the rtp and rtcp ports differ.



: Initialise of Client RTP sessions

### DoRender

We create a sink of the appropriate type. startPlaying is delivering the callback function afterPlaying which we need to extract our data out of FramedSource. Hence, we need to reimplement FramedSource according our needs to get the buffer. An easy correction for missing packages is it to swap the invalid current buffer with the previous buffer.

E.g.:

class myFramedSource : FramedSource {

public:

unsigned char \*getBuffer(){ return fTo; }

protected:

unsigned char\* fTo;

};

void afterPlaying(void\*) {

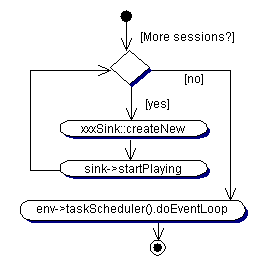
myFramedSource \*source = (myFramedSource \*)sink->source();

unsigned char\* buffer = source->getBuffer();

if (!buffer) buffer = previousBuffer;

Medium::close(sink);

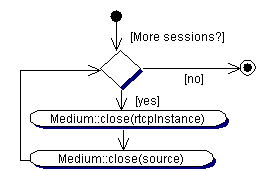
}



: DoRender for Client RTP sessions

### DeInitialise

We close rtcpInstance, sink and source for each sessionState. Closing the RTCPInstance will send a RTCP BYE.



: DeInitialise of Client RTP sessions

## Connector Module

# Outlook – next steps

After we have such a transfer connection system established we need to track the occupied bandwidth per connection, estimate the number of clients who might use the streaming concept and evaluate the quality of the data transfer. At a later stage we may add a bandwidth watchdog which is automatically adjusting the latency and muxing according the requested quality and the available bandwidth.

# Appendix

## Details about Application Layer

### Definitions

(Ref. RFC 3550, 3.)

**RTP payload**: The data transported by RTP in a packet […]

**RTP packet**: A data packet consisting of the fixed RTP header, a possibly empty list of contributing sources (see below), and the payload data. Some underlying protocols may require an encapsulation of the RTP packet to be defined.[…]

**RTCP packet**: A control packet consisting of a fixed header part similar to that of RTP data packets, followed by structured elements that vary depending upon the RTCP packet type. […]Typically, multiple RTCP packets are sent together as a compound RTCP packet in a single packet of the underlying protocol; […]

**Multimedia session**: A set of concurrent RTP sessions among a common group of participants. For example, a videoconference (which is a multimedia session) may contain an audio RTP session and a video RTP session.

**RTP session**: An association among a set of participants communicating with RTP. A participant may be involved in multiple RTP sessions at the same time. In a multimedia session, each medium is typically carried in a separate RTP session with its own RTCP packets unless the the encoding itself multiplexes multiple media into a single data stream. A participant distinguishes multiple RTP sessions by reception of different sessions using different pairs of destination transport addresses, where a pair of transport addresses comprises one network address plus a pair of ports for RTP and RTCP.

**Synchronization source (SSRC)**: The source of a stream of RTP packets, identified by a 32-bit numeric SSRC identifier carried in the RTP header so as not to be dependent upon the network address. All packets from a synchronization source form part of the same timing and sequence number space, so a receiver groups packets by synchronization source for playback.[…] The SSRC identifier is a randomly chosen value meant to be globally unique within a particular RTP session (see Section 8). A participant need not use the same SSRC identifier for all the RTP sessions in a multimedia session; the binding of the SSRC identifiers is provided through RTCP (see Section 6.5.1).

**Contributing source (CSRC)**: A source of a stream of RTP packets that has contributed to the combined stream produced by an RTP mixer (see below). The mixer inserts a list of the SSRC identifiers of the sources that contributed to the generation of a particular packet into the RTP header of that packet. This list is called the CSRC list.

**End system**: An application that generates the content to be sent in RTP packets and/or consumes the content of received RTP packets.

**Mixer**: An intermediate system that receives RTP packets from one or more sources, possibly changes the data format, combines the packets in some manner and then forwards a new RTP packet. Since the timing among multiple input sources will not generally be synchronized, the mixer will make timing adjustments among the streams and generate its own timing for the combined stream. Thus, all data packets originating from a mixer will be identified as having the mixer as their synchronization source.

**Translator**: An intermediate system that forwards RTP packets with their synchronization source identifier intact.

**Monitor**: An application that receives RTCP packets sent by participants in an RTP session, in particular the reception reports, and estimates the current quality of service for distribution monitoring, fault diagnosis and long-term statistics. The monitor function is likely to be built into the application(s) participating in the session, but may also be a separate application that does not otherwise participate and does not send or receive the RTP data packets (since they are on a separate port). These are called third-party monitors.

**Source-specific multicast** (**SSM**) is a method of delivering [multicast](https://en.wikipedia.org/wiki/Multicast) packets in which the only packets that are delivered to a receiver are those originating from a specific source address requested by the receiver. By so limiting the source, SSM reduces demands on the network and improves security.

### RTCP

(Ref: RFC 3550, 6.)

RTCP performs four functions:

1. Providing **feedback** **on the quality** of the data distribution
2. Carrying a persistent transport-level **identifier** for an RTP source called the canonical name or **CNAME**. Since the SSRC identifier may change if a conflict is discovered or a program is restarted, receivers require the CNAME to keep track of each participant. Receivers may also require the CNAME to associate multiple data streams from a given participant in a set of related RTP sessions, for example to synchronize audio and video. Inter-media synchronization also requires the NTP and RTP timestamps included in RTCP packets by data senders.
3. **Controlling rate** in order for RTP to scale up to a large number of participants.
4. Optional: convey minimal session control information, for example participant identification to be displayed in the user interface.

### RTSP

The Real Time Streaming Protocol, or RTSP, is an application-level protocol for control over the delivery of data with real-time properties. RTSP provides an extensible framework to enable controlled, on-demand delivery of real-time data, such as audio and video. Sources of data can include both live data feeds and stored clips. This protocol is intended to control multiple data delivery sessions, provide a means for choosing delivery channels such as UDP, multicast UDP and TCP, and provide a means for choosing delivery mechanisms based upon RTP.

The Real-Time Streaming Protocol (RTSP) establishes and controls either a single or several time-synchronized streams of continuous media such as audio and video. It does not typically deliver the continuous streams itself, although interleaving of the continuous media stream with the control stream is possible. In other words, RTSP acts as a "network remote control" for multimedia servers.

There is no notion of an RTSP connection; instead, a server maintains a session labeled by an identifier. An RTSP session is in no way tied to a transport-level connection.

The streams controlled by RTSP may use RTP, but the operation of RTSP does not depend on the transport mechanism used to carry continuous media. The protocol is intentionally similar in syntax and operation to HTTP/1.1. However, RTSP differs in a number of important aspects from HTTP:

* An RTSP server needs to **maintain state** by default in almost all cases, as opposed to the stateless nature of HTTP.
* Both an RTSP server and client can **issue requests**.
* Data is carried out-of-band by a different protocol. (There is an exception to this.)
* RTSP is defined to use ISO 10646 (UTF-8) rather than ISO 8859-1, consistent with current HTML internationalization efforts.
* The Request-URI always contains the **absolute URI**. Because of backward compatibility with a historical blunder, HTTP/1.1carries only the absolute path in the request and puts the host name in a separate header field.

The protocol supports the following operations:

* **Retrieval of media from media server**:

The client can request a presentation description via HTTP or some other method. If the presentation is being multicast, the presentation description contains the multicast addresses and ports to be used for the continuous media. If the presentation is to be sent only to the client via unicast, the client provides the destination for security reasons.

* I**nvitation of a media server to a conference**
* **Addition of media to an existing presentation**:  
  Particularly for live presentations, it is useful if the server can tell the client about additional media becoming available.

#### Terminology

* **Aggregate** **control**:  
  The control of the multiple streams using a single timeline by the server. For audio/video feeds, this means that the client may issue a single play or pause message to control both the audio and video feeds.
* **Conference**:  
  A multiparty, multimedia presentation, where "multi" implies greater than or equal to one.
* **Client**:  
  The client requests continuous media data from the media server.
* **Connection**:   
  A transport layer virtual circuit established between two programs for the purpose of communication.
* **Container file**:  
  A file which may contain multiple media streams which often comprise a presentation when played together. RTSP servers may offer aggregate control on these files, though the concept of a container file is not embedded in the protocol.
* **Continuous media**:  
  Data where there is a timing relationship between source and sink; that is, the sink must reproduce the timing relationship that existed at the source. The most common examples of continuous media are audio and motion video. Continuous media can be real-time (interactive), where there is a "tight" timing relationship between source and sink, or streaming (playback), where the relationship is less strict.
* **Entity**:  
  The information transferred as the payload of a request or response. An entity consists of metainformation in the form of entity-header fields and content in the form of an entity- body, as described in Section 8.
* **Media initialization**:  
  Datatype/codec specific initialization. This includes such things as clock rates, color tables, etc. Any transport-independent information which is required by a client for playback of a media stream occurs in the media initialization phase of stream setup.
* **Media parameter**:  
  Parameter specific to a media type that may be changed before or during stream playback.
* **Media server**:   
  The server providing playback or recording services for one or more media streams. Different media streams within a presentation may originate from different media servers. A media server may reside on the same or a different host as the web server the presentation is invoked from.
* **Media server indirection**:  
  Redirection of a media client to a different media server.
* **(Media) stream**:  
  A single media instance, e.g., an audio stream or a video stream as well as a single whiteboard or shared application group. When using RTP, a stream consists of all RTP and RTCP packets created by a source within an RTP session.
* **Message**:  
  The basic unit of RTSP communication, consisting of a structured sequence of octets and transmitted via a connection or a connectionless protocol.
* **Participant**:  
  Member of a conference. A participant may be a machine, e.g., a media record or playback server.
* **Presentation**:  
  A set of one or more streams presented to the client as a complete media feed, using a presentation description as defined below. In most cases in the RTSP context, this implies aggregate control of those streams, but does not have to.
* **Presentation description**:  
  A presentation description contains information about one or more media streams within a presentation, such as the set of encodings, network addresses and information about the content.
* **Response**:  
  An RTSP response. If an HTTP response is meant, that is indicated explicitly.
* **Request**:  
  An RTSP request. If an HTTP request is meant, that is indicated explicitly.
* **RTSP session**:  
  A complete RTSP "transaction", e.g., the viewing of a movie. A session typically consists of a client setting up a transport mechanism for the continuous media stream (SETUP), starting the stream with PLAY or RECORD, and closing the stream with TEARDOWN.
* **Transport initialization**:  
  The negotiation of transport information (e.g., port numbers, transport protocols) between the client and the server.