Design and Proof-of-Concept Implementation of Proxy-based Stream Handling for an Enterprise Service Bus

by

Tim Hedlund

LIU-IDA/LITH-EX-A--14/036--SE

2014-08-15
Final thesis

Design and Proof-of-Concept Implementation of Proxy-based Stream Handling for an Enterprise Service Bus

by

Tim Hedlund

LIU-IDA/LITH-EX-A--14/036--SE

2014-08-15

Supervisor: Vengatanathan Krishnamoorthi (IDA), Andreas Persson (Ida Infront)
Examiner: Niklas Carlsson
Abstract

Traffic for real-time entertainment applications has increased over the past years and now holds one of the largest portions of the traffic on the Internet. This trend has been spotted by Ida Infront who wishes to integrate and support a streaming protocol in their Enterprise Service Bus (ESB). This thesis first surveys three streaming protocols and techniques (SIP, RTSP and HTTP-base streaming) in terms of general usage areas and suitability for integration with the company ESB.

Second, this thesis investigates how these protocols can be implemented using a proxy-based design together with the ESB. After describing one design alternative for each protocol, the thesis finally presents a proof-of-concept implementation of an RTSP proxy. The proxy is implemented and tested for performance and added packet jitters under different loads.

The main contribution of the thesis is a reverse RTSP proxy application that integrates with the ESB and offers both routing and logging functionalities. In tests where we put the proxy under load we see that the CPU and memory usage is low and that the bottleneck appears to be the network bandwidth. But we also see a small added jitter that potentially could scale and must be taken into account when using such a proxy application.
## 4.3 Alternative 3 - HAS proxy

4.3.1 Scenario mapping

4.3.2 Record signaling

## 4.4 Comparison and choice

## 5 Proof-of-Concept Implementation

5.1 Stream handling and iipax

5.2 System overview

5.3 RTSP proxy

5.4 ESB communication

5.4.1 Routing

5.4.2 Management - Logging

5.5 RTP/RTCP tunnel threads

5.5.1 Recording

## 6 Results: Validation and Testing

6.1 Testing environment and setup

6.1.1 Machine P

6.1.2 Machine T

6.1.3 Machine J

6.1.4 Machine S

6.2 Performance test

6.3 RTP jitter test

6.4 Towards product integration

## 7 Related Work

7.1 Middleboxes

7.2 Proxy-based solutions

7.3 Streaming

## 8 Conclusions and Discussions

8.1 Summary

8.2 Conclusions

8.2.1 Survey

8.2.2 Implementation and ESB integration

8.2.3 Application testing

8.3 Discussion

8.3.1 Results

8.3.2 Method

8.3.3 Ethical aspects

8.4 Limitations

8.4.1 Survey

8.4.2 Proxy application

8.4.3 Testing

8.5 The future
Chapter 1

Introduction

1.1 Motivation

The data traffic on the Internet has in the past mostly been represented by web browsing and file sharing, but today this is no longer true. Instead, according to a recent study real-time entertainment traffic together with communication traffic as of 2013 was responsible for some 64.9 percent of the total aggregated download and upstream traffic in fixed access networks during peak period in North America [1].

With this global increase in streaming services and video streaming already being the dominant traffic type on the Internet, there is a growing interest among companies to investigate in this new trend. For example, companies such as Ida Infront wishes to explore what kind of stream protocols that is present today, what their main usage areas are, what type of functionalities that could be integrated with their Enterprise Service Bus (ESB), and how these types of services best can help serve their increasing demand for streaming services.

1.2 Problem specification

Today the ESB only supports message based communication over standard web services, but Ida Infront wishes to also support streaming protocols in the future. This thesis surveys three well known streaming protocols and techniques, and investigates how they could be integrated with the ESB.

To be able to understand the integration part and to give meaning to it a usage scenario is presented next. The scenario serves as an comparison point for the different protocols and to see how they can be used in the specific case.
1.2. General scenario

A general ESB scenario is described in Figure 1.1. In the figure there are multiple sinks (clients) that are communicating with sources (servers) through the ESB. The media clients could be multimedia players, Voice over Internet Protocol (VoIP) telephones, or general data stream receivers. The media servers could be surveillance cameras, multimedia servers, or another VoIP phone. The ESB already features a logging tool for messages which the company would like to use also for logging streaming sessions. Additionally, the scenario imposes the ability for the protocol to be able to signal to the ESB to record the stream.

Figure 1.1: Picture describing the scenario

1.2.2 Concrete workload example

Figure 1.2 shows a concrete example to clarify the scenario even more. This setup describes one media client who wants to receive a video stream from a media server. The client connects to the ESB and all requests are routed in real time to the media server. If a separate channel is used for the media stream, then that data also flows through the bus. If at some point the client sends a record command, that command has to be intercepted and trigger a recording process to mirror the stream to the storage.

1.2.3 Goals

The goals of this thesis is to extend the company ESB with support for a streaming protocol and to give insights into different streaming techniques, both in general and in a proxy context. The final output of this project is
an implemented proxy solution that integrates with the ESB and provides logging and routing functionalities for streams and also the ability to signal to the ESB to record the stream.

1.3 Contributions

This thesis makes three primary contributions to address the problem stated. First a system design is presented (Chapter 5), which is grounded based on a survey of existing protocols and techniques (Chapter 3) for the related problems (Chapter 2) and take into consideration different design alternatives (Chapter 4). The design is customized to address the challenges for the unique domain containing the stated scenario (outlined in Section 1.2).

Second, a proof-of-concept implementation of the system is implemented (Chapter 5). The proof-of-concept implementation shows how stream handling could be integrated with the company ESB. Also, a client interface is built for the company to show how the stream handling could possibly be used inside their iipax product suite, and to help illustrate the effectiveness and functionality of the solution.

Finally, a validation and performance evaluation of the final system are presented (Chapter 6), which allows us to discuss and provide insights into the value of the implementation and system design choices (Chapter 8).
Chapter 2

Background

2.1 Enterprise service bus

An Enterprise Service Bus (ESB) is an architecture for business integration which uses standard message based communication to decouple the services. In short, an ESB act as a communication bridge between different services. It also provides a way for the services to be choreographed to work in concert to accomplish more complex tasks.

![Figure 2.1: High level description of an ESB](image)

2.1.1 Core features

There is no general consensus in the industry regarding what an ESB must contain but there are some typical features that usually are included [2, 3]. Next is a list of typical features.
CHAPTER 2. BACKGROUND

Invocation
Standardized requests and responses sent over the bus, often done so with web service standard SOAP, WSDL and UDDI protocols.

Routing
Intelligent routing of messages both depending on address or on content.

Load Balancing
Mechanisms to distribute service loads over several providers.

Mediation
Translations between protocols or transformations of message data e.g. inch to centimeters.

Adapters
An ESB may provide adapters for connecting different softwares to the bus.

Security
Protection of the messages that are carried over the bus, i.e., message integrity.

Management
Tools for logging messages, traffic analysis and an administration point for the bus to setup, configure and control processes.

Process Orchestration
An engine that can combine different services into a more complex program.

Routing and logging of streams are two features that Ida Infront wishes to integrate with the ESB and both of them are discussed more in the design phase (Chapter 5) to see how they could be integrated into the solution.

2.1.2 iipax
The iipax system is a server/client Java Enterprise Edition (Java EE) application that can run on several different operating systems and supports several different database management systems and application servers. It is actually a suite of services ranging from communication and issue tracking to archive systems but at its core it is an enterprise service bus.

The bus is currently supporting protocols for electronic services such as SSEK [4] developed by the Swedish insurance mediators group (Svenska försäkringsförmedlarens förening), SHS (Spridnings- och Hämtnings system) [5] a protocol developed by Swedish authorities for secure distribution and gathering of data, and RIV (Regler för Interoperabilitet inom Vården) [6] a communication protocol developed by the Swedish health care.
2.2 Streaming protocols

When talking about streaming protocols there are often two approaches mentioned. First there is the standard web based streaming where web servers are used for the actual streaming. This is usually referred to as "web streaming". Secondly there are special protocols developed for the purpose of controlling and sending streams, such protocols uses special servers that implement some of the streaming protocols. But, there is more to this, the streaming protocols can also be divided into controlling protocols and transport protocols. To clarify this we take a look at Figure 2.2. Here we see that there are two separate channels for the control and data flow. By using this type of layered architecture several different control protocols can, and in fact do, use the same transport protocols.

This thesis explores the usage areas and functionalities of web streaming techniques (HTTP) and two protocols specifically built for streaming, the Session Initiation Protocol (SIP) and the Real Time Streaming Protocol (RTSP).

In order to fully understand the protocols explained in Chapter 3, the Real-time Transport Protocol (RTP) and its associated protocol the RTP Control Protocol (RTCP) are both explained in Section 2.3.

Figure 2.2: High level figure describing session control

Figure 2.3: Protocol overview of streaming scenario with RTP/RTCP.
2.3 RTP/RTCP

Streaming protocols often only provide the mechanism to setup and control the stream, not to send the actual streaming data. For that purpose they are instead typically used together with the Real-time Transport Protocol (RTP) RFC 3550 [7] which is a transport protocol specifically designed to carry real time data such as audio and video streams. The RTP protocol is not actually a transport layer protocol, it lives in the application layer (fig 2.3) and is therefore not dependent on the underlaying transport protocol but usually UDP is used to carry the RTP messages. The RTP Control Protocol (RTCP) RCF 3550 [7] is used together with RTP and its primary function is to provide feedback on the RTP stream.
Chapter 3

Protocol Survey and Mechanisms

This chapter covers the two open standards protocols, SIP and RTSP, plus the HTTP web techniques used in today’s web streaming. The survey will then be the basis for the system design in Chapter 5.

3.1 SIP

The Session Initiation Protocol (SIP) RFC 3261 [8] is an application layer protocol (Figure 3.1) that provides control (signaling) for creating, modifying and terminating sessions. It is widely used for (but not limited to) online telephony where a user is identified by a SIP address; e.g., sip:bob@194.12.123.10 or sip:bob@example.com. The SIP protocol can be transmitted over ordinary TCP, UDP, or over the Stream Control Transport Protocol (SCTP), which offers a combination of functionalities between TCP and UDP. As SIP itself is not designed to carry any voice or video, it is often used together with the RTP protocol.

3.1.1 Mechanisms

SIP mainly provides three mechanisms for handling sessions [9].

- One for setting up and establishing calls between two parties by letting them notify each other (ringing) and also to let them agree upon which media codecs that should be used.

- One which lets a caller look up the IP-address of a callee. SIP offers the functionality to its clients to move around in different networks and also to have multiple devices with different IP-addresses.
3.1.2 The SIP infrastructure

The SIP infrastructure is based on two server types. Proxy servers that receive and forward requests to other SIP proxies or to the requested client. Registrar servers that every user agent is associated with and are responsible for register mappings between user agents and their current IP-addresses into a location service. The location service is used by the proxy for forwarding or lookup.

To show how this works in practice, let’s take a look at Figure 3.2 below. Here Bob wants to talk to Alice without knowing her IP-address. Whenever a SIP application is started it needs to register itself with its registrar, so Alice’s user agent has probably already sent a REGISTER message to her registrar server telling what IP-address her device has. (1-2) The first thing Bob does is to send an INVITE message to Alice on her SIP-address “sip:alice@example.com” through his proxy server. (3) Alice’s proxy server then receives the message and makes a lookup in the location service. It can now forward the message to the IP-number of the SIP client currently registered. (4-6) Alice now sends a SIP response through the proxies and (7) the two parties can finally send media directly between the devices.

3.1.3 SIP messages

SIP messages are either requests or response messages. The syntax of the message is very much alike the HTTP/1.1 syntax, though it differs in character set and syntax specifics.

\[
generic-message = start-line *message-header
\]
3.1. SIP

Figure 3.2: Example for showing the SIP infrastructure inspired by RFC 3261 [8].

CRLF

[ message-body ]

start-line = Request-Line / Status-Line

This thesis will not explain the protocol syntax in any greater depth but the interested reader is encouraged to read RFC 3261 [8], RFC 2822 [10] and the HTTP/1.1 specification RFC 2616 [11]. Instead we take a look at the INVITE message and the response from the previous example. The INVITE method is used to signal to an endpoint that it wants to initiate a session. Figure 3.3 illustrates the message exchange on a more detailed level. Bob

Figure 3.3: The SIP INVITE flow.
sends a INVITE message to his SIP proxy:

```
INVITE sip:bob@idainfront SIP/2.0
Via: SIP/2.0/UDP scorch.idainfront.se
     ;branch=z9hG4bK776asdhds
Max-Forwards: 70
To: Alice <sip:alice@example.com>
From: Bob <sip:bob@idainfront.com>;tag=1928301774
Call-ID: a84b4c76e66710@scorch.idainfront.se
CSeq: 1337 INVITE
Contact: <sip:bob@scorch.idainfront.se>
Content-Type: application/sdp
Content-Length: 142

[SDP not shown]
```

The message request-line contains the INVITE, URI and the version code. The Via header is used for response routing, the address tells where Bob is expecting to receive the response. When the INVITE message passes any middle point a new via header is added. The headers To, From and Call-ID together make up a globally unique identifier for the call. CSeq is an integer marker for each request and is incremented for every new request. The Contact header provides a direct SIP URI where future requests could be sent to. Finally the payload of the message contains a Session Description Protocol (SDP) RFC 4566 [12] message. The SDP message itself contains parameters for the stream initialization.

The INVITE message is then forwarded all the way to Alice’s SIP phone and while the message is forwarded, the proxy servers answers back with a "100 Trying" response to indicate that the INVITE was received and is processed. Alice’s SIP phone is then responding with a "180 Ringing" message indicating that the phone is ringing. When Alice decides to answer her phone, it will send back a "200 OK" message.

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP proxy.example.com
     ;branch=z9hG4bKnasahds8;received=192.0.2.3
Via: SIP/2.0/UDP proxy.lkpg.idainfront.se
     ;branch=z9hG4bK77ef4c2312983.1;received=192.0.2.2
Via: SIP/2.0/UDP scorch.idainfront.com
     ;branch=z9hG4bK776asdhds ;received=192.0.2.1
To: Alice <sip:alice@example.com>;tag=a6c85cf
From: Bob <sip:bob@idainfront.se>;tag=1928301774
Call-ID: a84b4c76e66710@scorch.idainfront.com
CSeq: 1337 INVITE
Contact: <sip:alice@192.0.2.4>
Content-Type: application/sdp
Content-Length: 131
```
The first line contains the SIP version, status code and a status string describing the response. We can also see that Via headers have been added on the way to Alice. Alice also provides her SDP message in the response.

After Bob has received the response from Alice he sends an ACK message back to Alice directly through the SIP URI learned from the Contact header in the response message. This is done to acknowledge that he got the response and agreed upon the session SDP parameters. This procedure is called the INVITE/200/ACK three-way handshake. Now both parts are ready to start talking to each other over a separate channel that was agreed upon in the SDP messages. This separate channel would normally use the RTP protocol for transporting real-time data.

When Alice and Bob are finished talking, Bob presses the end call button in his softphone which sends a BYE message to Alice. Alice’s phone then answers with an 200 OK message and the session is terminated.

3.2 RTSP

The Real Time Streaming Protocol (RTSP) RFC 2326 [13] is a popular control protocol for delivery of real-time data. It offers on-demand stream establishment and control for audio, video and conferencing. RTSP can at a high level be seen as a remote control, with commands for SETUP, PLAY, PAUSE and TEARDOWN. Additionally RTSP has a RECORD method for server stream recording. Figure 3.4 shows RTSP in the Internet protocol stack. RTSP is also an application layer protocol like the Session Initiation Protocol and it can be transmitted using both the reliable data transfer offered by TCP or the message oriented transfer by UDP. RTSP is like SIP not designed to carry any voice or video. Usually the RTP protocol is used in combination to carry the actual media data. Although it should be said that RTSP can send streams interleaved over TCP, but it is strongly discouraged because of the additional overhead [13].

3.2.1 Mechanisms

RTSP does not offer any end point lookup or ringing mechanism like the Session Initiation Protocol does. It is merely an end to end client to server protocol that provides two important mechanisms for session handling.

- First, RTSP offers session initiation and establishment between a client and a server. This is done by an agreement between the server and client that states what protocols and transport channels that will be used for the stream.
• Second, RTSP offers session control in a connectionless manner. PLAY, PAUSE etc can be sent from any client that knows of or is receiving a stream.

### 3.2.2 RTSP messages

The RTSP message syntax is like SIP very much alike the HTTP syntax and follows the request-response paradigm. It consists out of a start-line, followed by message headers and finally the message body.

\[
generic-message = \text{start-line} \\
\quad \text{*message-header} \\
\quad \text{CRLF} \\
\quad [ \text{message-body} ] \\
\text{start-line} = \text{Request-Line / Status-Line}
\]

The interested readers are encouraged to read more in-depth about the syntax and semantics in the RFC 2326 [13].

To explain the message flows and the general usage of the RTSP we take a look at Figure 3.5. We consider a basic scenario where the client wants to receive a video stream from the media server. The client first sends a message using the OPTIONS method to receive a list of all the possible commands understood by the server.

```
OPTIONS rtsp://idainfront.se/gp.ts RTSP/1.0
User-Agent: LibVLC/2.1.3 (LIVE555 Streaming Media v2014.01.21)
CSeq: 2
```

The URL is pointing to the resource “gp.ts” (which is a MPEG transport stream container format). In RTSP, the URL always contains the absolute path contrary to HTTP. The CSeq specifies the sequence number for a RTSP request- response pair.
The server now answers the request with a "200 OK" response containing a list of valid commands.

Figure 3.5: A common RTSP client and server interaction for session setup, play and teardown.

RTSP/1.0 200 OK
Date: Wed, Mar 05 2014 08:49:10 GMT
CSeq: 2
Public: OPTIONS, DESCRIBE, SETUP, TEARDOWN, ;PLAY, PAUSE, GET_PARAMETER, SET_PARAMETER

After the client received the command list, it sends a DESCRIBE message.

DESCRIBE rtsp://idainfront.se/gp.ts RTSP/1.0
User-Agent: LibVLC/2.1.3 (LIVE555 Streaming Media v2014.01.21)
Accept: application/sdp
CSeq: 3

The DESCRIBE method retrieves a description of the media object defined in the request URL. The description protocol is independent from RTSP but normally SDP (discussed in 3.1.3) is used.
RTSP/1.0 200 OK
Content-Base: rtsp://idainfront.se/gp.ts/
Date: Wed, Mar 05 2014 08:49:10 GMT
CSeq: 3
Content-Type: application/sdp
Content-Length: 390

[SDP not shown]

Now the client is ready to setup a stream session so it send a SETUP message with a transport header. The header contains information of the preferred transportation protocol, cast type and port numbers.

SETUP rtsp://idainfront.se/gp.ts/track1 RTSP/1.0
User-Agent: LibVLC/2.1.3 (LIVE555 Streaming Media v2014.01.21)
CSeq: 4
Transport: RTP/AVP;unicast;client_port=61590-61591

The server confirms the setup with a "200 OK" message. The response contains a transport header which echoes the requested header plus source and destination addresses and server port numbers. It also contains a session header which is a RTSP session identifier that needs to be included in any session control message for the current session.

RTSP/1.0 200 OK
Date: Wed, Mar 05 2014 08:49:10 GMT
Session: C8B48B68;timeout=65
CSeq: 4
Transport: RTP/AVP;unicast;destination=192.168.40.123;source=127.0.0.1;client_port=61590-61591;server_port=42738-42739

After the setup, the client and server are both ready to start streaming data on the specified ports. To start the stream, the client sends a PLAY message. The message includes a range header stating from which time of the stream it wants to start receiving.

PLAY rtsp://idainfront.se/gp.ts/ RTSP/1.0
User-Agent: LibVLC/2.1.3 (LIVE555 Streaming Media v2014.01.21)
Session: C8B48B68
CSeq: 5
Range: npt=0.000-

After a while when the stream has played to the end or the user has pressed the stop button in the client, the client sends a TEARDOWN message indicating that it wants to stop the streaming session.
The server acknowledges the teardown with a "200 OK" and the session and stream are terminated. This concludes the RTSP session establishment, control and teardown example.

3.3 HTTP

This section is not explicitly covering any of the HTTP message syntax or semantics. Instead the focus lies on HTTP functionalities in a streaming context. The truth is that today the major part of the streaming traffic comes from big on-demand stream providers like Netflix and YouTube which uses HTTP-based streaming techniques [1, 9].

HTTP has two distinct advantages over other protocols built for streaming. The first is because of that ordinary web servers can provide the content, companies which already have web server infrastructures already got the streaming back-end for "free".

The second is that firewalls often are configured to let web traffic pass through. This powerful combination is a big advantage for companies wanting to stream data to customers on the Internet [14].

There are two HTTP techniques used for enabling media playback streams: progressive download and HTTP-based Adaptive Streaming.

3.3.1 Progressive download

In the past a web browser had to download the entire media file before it could start the playback, meaning that a user had to wait the entire download time before consuming the media. But then came progressive download which also downloads the media file, but differs in that the browsers can begin the playback as the file is being downloaded. This means that if the media file consumption rate is lower then the download rate, the file buffer will grow, possibly bigger than necessary if the user chooses to stop the playback before the end of the file, and thus wasting bandwidth. On the other hand if the consumption rate instead is higher than the download rate the playback will stop when the local buffer runs out. This problem is solved by the adaptive bitrate streaming approach.

3.3.2 HTTP-based Adaptive Streaming (HAS)

With HTTP-based Adaptive Streaming (HAS), the media files are split into chunks, and each chunk is available in different qualities and playback rates. This means that different qualities of the stream can be received and played
Figure 3.6: The adaptive bitrate stream where the clients start to receive high quality pieces of a video file but suddenly change to receive a lower quality, possibly by noticing network congestion.

depending on the current connection bandwidth. Each bitrate media versions is normally divided into short segments a few seconds long. So if the connection capacity changes in real time the stream can adapt thereafter and receive a higher or lower bitrate version of the media file as shown in Figure 3.6.

There are many proprietary implementations available today where Adobe’s HTTP Dynamic Streaming (HDS), Apple’s HTTP Live Streaming (HLS) and Microsoft’s HTTP Smooth Streaming (HSS) are among the biggest. But an open standard MPEG-Dynamic Adaptive Streaming over HTTP (MPEG-DASH) is also under development.
Chapter 4

Design Alternatives and Selection

SIP, RTSP and HTTP have all previously been surveyed in terms of general usability and infrastructures. In this chapter we consider three proxy design alternatives, one for each protocol. All alternatives are discussed in terms of how a proxy solution can solve the problems in the general scenario (Section 1.2.1), what needs to be modified in the proxy to make the streams to pass through it and also how the protocols could support record signaling. For the HTTP protocol proxy, HTTP-based Adaptive Streaming is considered as it is the most common content delivery system today [1].

The alternatives proposed in this section is kept at a very high level and is done so on purpose to limit the complexity. The alternatives are only suggestions and is the basis for the more complex system described in the proof-of-concept implementation chapter.

4.1 Alternative 1 - SIP proxy

SIP comes with a big infrastructure that demand a registrar server, a proxy server and a lookup service. The SIP proxy as explained in the survey chapter routes requests to the user’s current location and does not transfer any voice data streams. Therefor the SIP Proxy needs to be modified so that it also mediates the actual data streams. The mediation could be achieve by modifying the SDP message inside the INVITE messages. To be able to trick the two parties into sending their voice (RTP) data streams trough the SIP proxy we need to manipulate the SDP message contained in the INVITE message. We need to change the connection address inside the connection field and the port number inside the media field to that of the proxy instead to trick the clients. Next we see the connection field (c) and the media field (m) inside the SDP protocol (first discussed in Section 3.1.3).
CHAPTER 4. DESIGN ALTERNATIVES AND SELECTION

They typically have the following structures.

c=<network type> <address type> <connection address>
m=<media> <port> <transport> <fmt list>

illustrating the address and ports that must be modified.

4.1.1 Scenario mapping

When mapping SIP to the scenario in 1.2.1, the sources and sinks can, for example, be changed to two SIP user agent clients (UAC). The UAC’s could make a call through the proxy and the call could be recorded by either part by signaling to the ESB to record.

4.1.2 Record signaling

One problem with the signaling of the record command is that there is no clear way of doing so with SIP as there is no request method that could be used directly. But, one way is to use the SIP Message Extension defined in RFC 3428 [15] to signal. The extension allows for instant messages to be sent over the SIP network and this could probably be used to formulate a record command.

4.2 Alternative 2 - RTSP proxy

The RTSP protocol is very lightweight in that no infrastructure in between end-points is needed so a proxy solution similar to the SIP design could also be used here. Although, the RTSP protocol differs a bit from how SIP lets its user agents agree upon how to communicate. RTSP sets up the stream through its own header fields as we saw in the survey chapter. To trick the client and server to send the stream through the RTSP proxy we need to change the port numbers inside the transport header inside the client SETUP request/response to and from the server. Next is the transport header inside the client SETUP request. The transport header contains information about how the client wants to receive the stream.

SETUP rtsp://idainfront.se/gp.ts/track1 RTSP/1.0
User-Agent: LibVLC/2.1.3 (LIVE555 Streaming Media v2014.01.21)
CSeq: 4
Transport: RTP/AVP;unicast;client_port=61590-61591

4.2.1 Scenario mapping

To map the RTSP design to the scenario, the source is switched to a media server and the client is switched to a media client, e.g., a LIVE555 Media Server and a VLC media player. The client could request a live video stream through the proxy and could be signaled to be recorded.
4.2.2 Record signaling

The RTSP command RECORD can be used to signal to the proxy to record the data stream and thus fulfills the record part.

4.3 Alternative 3 - HAS proxy

The final alternative is to use a HTTP-based adaptive stream proxy. By using such a proxy we do not need to do any specific modification the messages to get the stream to go through the proxy as the stream is sent over the same connection. But as the HAS approach uses several small pieces to make up a longer movie the logging of streams becomes a problem. Do we want to log each small piece or the movie in full?

4.3.1 Scenario mapping

The web streaming approach maps to the scenario by placing a standard web server as the source and a browser client that supports HAS as the sink. The browser client could request chunks through the proxy and also signal to let them be recorded.

4.3.2 Record signaling

The record message can be a simple PUT or POST request to the same URL as the file that is being streamed.

4.4 Comparison and choice

It is difficult to compare the different designs and protocols as it all comes down to what usage area one wants to support and the scenario is not giving any requirement on that. But, one could summarize the different designs in that if one is looking for a mobile solution where end points are moving around much, the SIP protocol would be a good choice. If one instead wants to stream many live feeds and do not want a big infrastructure, the RTSP protocol can come in handy. Finally, if one would like to make use of existing web infrastructure to stream audio and video through any firewall with quality adaption the HAS alternative would be a good choice.

As mentioned above, each of the candidate designs have their advantages and disadvantages. However, for the relatively general example scenario discussed in this thesis there is no clear winner or dominating factor that must be taken into consideration. For the purpose of our proof-of-concept implementation of the general service outlined in this thesis, we instead relatively arbitrarily pick the RTSP design. With available media players and servers
supporting RTSP, the choice of RTSP also allowed fast implementation, allowing the focus to be placed on the high-level design and the performance of the system.

It should also be noted that the UDP-based solutions are expected to work well within enterprise scenarios (as enterprises usually have control of their own firewall settings) and UDP-based traffic is being on the upswing over the last few years [16].
Chapter 5

Proof-of-Concept Implementation

The design alternative presented in the RTSP proxy proposal did not show how the proxy interacted with the ESB, i.e. iipax. This was done purposely to keep the design more general and to not force any implementation specific restraints to how the proxy can be integrated with the ESB, it could be built into the ESB or built as a standalone application.

The proof-of-concept implementation, however, is built as a standalone Java Standard Edition (Java SE) application that communicates with the iipax application over Remote Method Invocation (RMI). Why a standalone RTSP proxy was chosen was because of iipax did not allow to open new port for communication other than over HTTP and we need one TCP port for RTSP and two UDP ports for RTP/RTCP to be able to stream one single stream.

5.1 Stream handling and iipax

Two features that play a central role in the iipax system is Routing and Management so focus have been on implementing support for them and integrating them with iipax.

Iipax already offers built in support for them and the routing is available by using a built in rule table in the iipax system. The rule table can be modified so that a RTSP url can be mapped to a IP-address. The proxy uses the rule table to figure out which server that is hosting the resource given in the RTSP url, e.g. "rtsp://host/resource.ts", and then connects to it and forwards the requests.
5.2 System overview

An overview of the system is shown in Figure 5.1, where we can see how all parts communicate with each other. The client requests a stream over RTSP through the proxy which in turn communicates with iipax over RMI calls. The actual stream is then traveling from the media server to the media client through the proxy over RTP. RTCP is also traveling the same way but is not shown in the figure. The proxy is also connected to a storage in the figure but in the implementation it is only a path to some location where the recordings are stored.

Figure 5.2 shows a more detailed system overview, including the different modules that makes up the proxy application. The proxy is mainly made up of three parts, namely a "ESB communication" part that handles all communications with iipax, a "RTSP Proxy" part that takes care of all RTSP communications between the media client and the media server, and finally a "RTP/RTCP Proxy" part that manages the forwarding of RTP and RTCP packets between the media client and the media server. It also handles the actual recording of the stream. The figure also shows the ports on which the proxy communicates on during a streaming session. The three modules are explained fully in the following sections.
5.3 RTSP proxy

The proxy module is the part that takes care of all RTSP messages that travel through the proxy. For every new RTSP (TCP) connection to port 8989 a new proxy thread is created and started to handle that session.

Figure 5.3 shows the main request/response loop in the RTSP proxy. The flow starts when a new connection is established from a client, the proxy waits for a request from the client, it reads the message and make an address lookup through the ESB communication module and connects to the media server if it is not already connected. Next, the message gets logged by the ESB communication module to iipax if it contains one of the request methods that is configured to be logged. Then the client’s message get transformed into a message that appears to the media server that as if it comes from the proxy and not from the client. The response from the media server is likewise transformed to appear to the client as if it came from the proxy. The transformation of the messages is important to be able to make the stream flow through the proxy and not directly between the media client and the media server. (The transformation is explained in further detail in Section 5.5). The loop goes on until any of the parties close the connection to the proxy.

5.4 ESB communication

The ESB communication module is the integration link between the proxy and iipax. It handles both the routing and the logging of sessions and messages, and does so over RMI calls.
5.4.1 Routing

Routing is done by the proxy when the first request is made by the client. The client requests a resource, e.g.
5.4. ESB COMMUNICATION

OPTIONS rtsp://host/resource.ts RTSP/1.0
User-Agent: LibVLC/2.1.3 (LIVE555 Streaming Media v2014.01.21)
CSeq: 2

The url is then sent to iipax which first makes a lookup to map the url to a unique product. The product is a term used in the iipax system for describing each message. Each message that is sent over the bus thus relates to a product-type. In our streaming case, each product-type corresponds to a stream service. So after the first lookup, a second lookup is made to iipax to receive an IP-address and a port to connect to.

Figures 5.4 and 5.5 show the rule table administration interface in iipax for both lookup tables. In Figure 5.4 we see the IIP-Product table where URL is associated with a product-type. Figure 5.5 shows the table where a product-type gets mapped to an address and port to a media server or possibly another ESB.

![Figure 5.4: The URL to product-type table.](image)

![Figure 5.5: The product-type to address table.](image)

26
5.4.2 Management - Logging

Each RTSP session is logged in the iipax system. iipax message logging system was build for the SHS protocol and each message can have several different statuses e.g. "Created", "Accepted", "Validated" and many more. But, in our case we do not have just single messages but instead streaming sessions. Therefore each message is instead considered as a stream session in the logging system and uses a subset of the statuses. The different statuses are corresponding to the RTSP session states, as shown below in Table 5.1. The RECORD command is not yet implemented.

<table>
<thead>
<tr>
<th>RTSP status</th>
<th>Message status</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETUP</td>
<td>Accepted</td>
<td>Yellow</td>
</tr>
<tr>
<td>PLAY</td>
<td>Processing</td>
<td>Yellow</td>
</tr>
<tr>
<td>TEARDOWN</td>
<td>Delivered</td>
<td>Green</td>
</tr>
<tr>
<td>RECORD</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

Table 5.1: The mapping between RTSP status, iipax message system status and color code.

Each stream session gets logged into the iipax system and can be viewed through a web interface as showed in Figure 5.6. The background color of the message rows changes depending on their current statuses. In the screen-shot we see how every stream gets logged into the system with an unique ID, Direction, Status, From, To, Product-type, Transaction ID and when started and stopped.

Figure 5.6: The message log in the iipax admin web interface
5.5 RTP/RTCP tunnel threads

The proxy spawns for each session a pair of tunnel threads that handles the forwarding of all RTP/RTCP traffic over UDP. The port numbers on which to listen and send to are extracted from the SETUP request and response messages. The port numbers in the request and response are changed so that the traffic flows through the tunnels. Figure 5.7 shows how the proxy manipulates the port numbers in the RTSP messages.

![Figure 5.7: Message diagram showing how the proxy changes the port numbers in the communication between client and server.](image)

5.5.1 Recording

The recording functionality is supported by signaling but no explicit recording code has been implemented to handle the actual recording of the RTP stream. To be able to record the stream, each RTP packet must be read and interpreted properly, which is outside the scope of this thesis.
Chapter 6

Results: Validation and Testing

To be convinced that the proxy application is actually working under load, some testing is necessary. Testing is important to discover both bottlenecks and possible limitations of the solution.

The proxy application is first tested for system load, including CPU usage, memory usage, and network bandwidth usage. Second, the quality of a RTP stream is tested for added jitter\(^1\) while using the proxy application and while not using it.

6.1 Testing environment and setup

The tests were conducted in a local area network with 1 Gbit switches as shown in Figure 6.1. There is no underlying thoughts about the position of the machines. The proxy application and the ipax system were installed on machine P. (The machine specifications are listed in Table 6.1.) Machine S is the media server which hosts the media files. Finally, machine T and J are test machines.

<table>
<thead>
<tr>
<th>Machine</th>
<th>OS</th>
<th>CPU</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>Win 7</td>
<td>8-Core i7-2600 CPU @ 3,80 GHz</td>
<td>16,0 GB</td>
</tr>
<tr>
<td>T</td>
<td>Win 7</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>J</td>
<td>Win 7</td>
<td>4-Core Q8200 @ 2,33 Ghz</td>
<td>4,00 GB</td>
</tr>
<tr>
<td>S</td>
<td>Win 2008 R2</td>
<td>4-Core E5-2630 @ 2,30 GHz</td>
<td>8,00 GB</td>
</tr>
</tbody>
</table>

Table 6.1: Machine specifications.

\(^1\)The variation in packet delay is generally referred to as jitter.
6.1. TESTING ENVIRONMENT AND SETUP

Figure 6.1: The test environment topology.

6.1.1 Machine P

The proxy application runs on this machine and receives incoming RTSP connections on port 8989. This machine also hosts an iipax communication installation. The iipax runs on a jboss4 application server and uses MySQL as database management system.

To be able to make measurements on the proxy application’s system load we use YourKit Java Profiler 2013. The profiler is hooked up to the proxy application from Eclipse. The profiler gives a lot of information about the application such as CPU/Memory usage and much more.

6.1.2 Machine T

This machine runs VLC clients that connects to the proxy (Machine P) and receive streams. To be able to spawn many VLC clients at once a batch script is used. Here is a script that starts 30 streams with no video/audio playback:

```
@ECHO OFF
for /l %%x in (1,1,30) do (  
    echo %%x
    start vlc --play-and-exit --novideo --noaudio rtsp://scorch:8989/happy.ts
    ping 192.168.40.254 -n 1 -w 200 > nul
)
```

6.1.3 Machine J

This machine is used to test video jitter. A single stream is analyzed with Wireshark’s built in RTP Stream analysis tool for jitter and packet losses.

6.1.4 Machine S

The server machine runs a slightly modified RTSP LIVE555 Media Server. It had to be modified so that more instances of the server could be started.
CHAPTER 6. RESULTS: VALIDATION AND TESTING

Originally it only supported two instances, each with the capacity to serve 32 simultaneous streams. The modification enabled it to open 8 instances; i.e., 256 simultaneous streams.

6.2 Performance test

To test the performance of the proxy application, the system was put under different system loads. We performed experiments with different number of concurrent clients streaming a video file through the proxy simultaneously. Each video stream had a bitrate of about 995 Kbit/s.

The results were captured with the profiler and are shown in Figures 6.2 and 6.3. Figure 6.2 shows the max and average CPU and bandwidth utilization for 30, 60, 120 and 240 concurrent streams. Figure 6.3 shows heap and non heap usage by the proxy under the same system loads as for CPU and bandwidth. Based on the results we see that it is obvious, looking at the bandwidth utilization for different loads that the bandwidth is the bottleneck when the number of streams increases. The bandwidth usage increases while the CPU and memory usage stays relatively low.

Due to the system performance nature of this test it is still difficult to say much about how well the proxy handles the streams in terms of video and audio quality in this test. However one can say that the system load is acceptable in terms of CPU and memory usage for up to 240 simultaneous 995 Kbit/s streams. These results are encouraging!

Figure 6.2: CPU and network utilization during different system loads.

6.3 RTP jitter test

From the performance test not much can be said about the video stream quality observed by the player and in the end by the user. To understand
6.3. RTP JITTER TEST

how the proxy application effects the stream we instead tested the RTP stream quality through measuring the added jitter.

First the jitter was measured without the interference of the proxy. In the same way as the performance testing, we tested 30, 60, 120 and 240 simultaneous streams to load the media server, and for each load we started up one more stream on machine J and measured the RTP jitter with Wireshark. After that, the same measurements were made when using the proxy application.

The results are shown in Figures 6.4 and 6.5. Figure 6.4 shows how the max jitter variates depending on system load and Figure 6.5 shows how the average jitter variates depending on system load. From the results in the Figures we see that when we do not use the proxy both the max and mean jitter is kept low and the mean jitter is nearly constant for the different loads. The mean jitter is kept under half a millisecond and the peak or max jitter average is about just under four milliseconds. When we instead look at the measurements with the proxy in use we see that there is clear that jitter is added, both the mean and the max.

Added jitter can cause poor video and audio playbacks, this problem is often almost eliminated by media players by the use of jitter buffers [9]. A jitter buffer on the other hand can increase the latency and thus might effect the users live stream experiences.

The mean jitter added by the proxy is still relatively small for 240 simultaneous streams. This relatively small average jitter addition of 3.5ms can be considered okay. However, without experiments with even more simultaneous clients or use in less ideal non-lab environments, it is unclear if the proxy eventually may cause real time latency problems. It it, however, encouraging that the maximum jitter increases much less as the load is increasing.
CHAPTER 6. RESULTS: VALIDATION AND TESTING

6.4 Towards product integration

To give the proxy application a front-end and improve the usability of our solution, a new workspace called xStream was build into the iipax client application. A screen shot of the workspace can be seen in Figure 6.6, the streams (products) are listed to the left and a media player sits to the right. xStream allows the user to utilize the streaming abilities of the newly integrated RTSP proxy and adds value to the iipax client.

The workspace is build as a Java Server Pages (JSP) generated HTML page that is viewed through an Internet Explorer embedded view port inside the iipax client. The .jsp file is hosted via a java servlet that runs inside the same application server as the iipax server.

The streams are dynamically loaded via RMI calls from the iipax back-end in the same way as the proxy application. The media player is a Chimera Web Plugin player that uses the VLC library, and can be controlled by the
buttons underneath it.

Figure 6.6: The xStream workspace inside the iipax client.
Chapter 7

Related Work

7.1 Middleboxes

The proxy application implemented in this thesis is a middlebox according to the Internet Engineering Task Force’s (IETS) definition of a middlebox [17]: “A middlebox is defined as any intermediary device performing functions other than the normal, standard functions of an IP router on the datagram path between a source host and destination host.”

Sherry et al. [18] argues that middleboxes come with high infrastructure and management costs and that enterprises could benefit from outsourcing them to the cloud. Using measurements on their own implementation, the Appliance for Outsourcing Middleboxes (APLOMB) they found that APLOMB imposes only a small average latency increase (approximately 1ms) and a median bandwidth inflation (approximately 3.8%).

Other research [19] has shown that by removing middleboxes and pushing them to trusted zones at end hosts inside the enterprise can give administrators more control at a lower cost. Sekar et al. [20] highlights the opportunities and challenges that arises in a middlebox context and proposes a consolidated, software-centric architecture for middleboxes in enterprise networks.

7.2 Proxy-based solutions

There are many HTTP proxys in the web today and they play an important role in both security and service optimization for companies and organizations. The proxies caches data to minimize the delays. But when applying proxies to HTTP-based adaptive streaming techniques where clients have an important role to decide what bitrate quality that will be fetched next problems might arise. Krisnamoorthi et al. [21] shows how the media client might get fooled by the speed of which the proxy delivers content and thus next requests a possibly to big bitrate version for the proxy-to-server link.
to handle. They therefore propose an alternative approach where the client and proxy share information to improve the proxy performance. Other research on HAS-based proxies includes segment-based streaming media proxies. Chen et al. [22, 23] proposes techniques for maximizing cache utilization and prefetching for continuous media delivery.

RealPlayer also provides some technical documentation [24] on how to use the RTSP protocol with firewalls, proxies and other middleboxes, which was valuable for the RTSP proxy design in our solution.

Finally, it should be noted that proxy performance also has been studied for regular web content. This includes effective caching policies [25] and impact of hierarchical caches [26].

### 7.3 Streaming

Yong et al. [27] proposes an approach to design and implement a Performance Testing Utility for an RTSP Streaming Server. They suggest testing the server for ”Maximum number of concurrent streams”, ”Aggregate bandwidth”, ”Packet loss rate” and ”CPU utilization”. In our test we do not look at packet losses but instead measure the added jitter.

Much research has been made on HAS-based streaming [14, 28, 29] and its benefits. Most of this research have ignored the characteristics of the video itself. In contrast, Li et al. [30] propose an optimization to the conventional HAS-streaming that takes the natural variations in the video into consideration. Integrating HAS and some of these ideas into the HTTP design alternative discussed in Section 4.3 could improve the resource utilization and provide continuous playback at the client. This may be an interesting direction for future work.
Chapter 8

Conclusions and Discussions

8.1 Summary

The thesis has answered questions about how the three streaming techniques, RTSP, SIP and HTTP-based streaming works and what their main usage areas are in a survey. We have seen that they all got different benefits and drawbacks depending on usage areas and prerequisites.

After the protocol survey one proxy solution for each protocol was discussed in terms of how it could fulfill and solve the initial problem given in the introduction. The RTSP protocol was then chosen to be implemented and integrated with iipax as a proof-of-concept implementation. Two features were recognized and integrated with the iipax, namely routing and logging of streams.

The main contribution, a reverse proxy application for the RTSP protocol was then tested for performance. The performance results provides insights about how well the proxy application performed under different loads.

Finally, a client interface within the iipax client were constructed to display RTSP streams to users. The interface demonstrates how the new stream handling functionality could be integrated into the iipax client application.

8.2 Conclusions

8.2.1 Survey

Looking back at the protocols and techniques in the survey, we see that they all have different pros and cons. The HTTP-based techniques benefits from its abilities to penetrate firewalls and the possibility to utilize existing
frameworks and infrastructures. But on the other hand they suffer from added delays and fairness policies from the TCP protocol [9].

The SIP protocol opens up for multiple and moving devices to connect, and offers flexibility in that it could be used together with different stream protocols even though it’s mainly used in telephone and messaging applications. On the downside the SIP protocol comes with quite a big and cumbersome infrastructure that needs to be installed and configured in order to work.

The final candidate in the survey was the RTSP protocol that also was chosen for implementation later on. As said, the choice were made relatively random because of that the company did not have any specific wishes to pick anyone over the others as they all seemed to work for the given scenario. The RTSP protocol was however picked, and some of the benefits that were recognized was the end to end-point nature that made it relatively lightweight compared to the SIP protocol and at the same time offered real time streaming abilities over UDP with a nice set of message methods compared to the HTTP-base variants. This together with the access to existing media clients and servers was making RTSP a good choice.

### 8.2.2 Implementation and ESB integration

The implementation ended up as a stand alone proxy application that integrated with the iipax via RMI calls. As said earlier, the application was made stand alone because of that iipax did not allow for new incoming connections other than over HTTP. However, the proxy application integrated with two features of the iipax system, routing and the logging management system. The routing is key to be able to change media server for any stream product in the system. The logging system logs every new stream session and keeps track of the current statuses of the streams.

From the beginning there was a request from the scenario that the proxy should be able to record the streams, but this have shown to be a bit difficult as the proxy need to be able to handle and read RTP streams as-well to be able to reconstruct the recording. As for now this has been left out to future work and implementations.

### 8.2.3 Application testing

Two tests were constructed to at least give a hint of how well the proxy performed under different loads. The first one was a system performance test to see how much resources the proxy application was allocated from the operating system. The test showed that the proxy handled the streams quite well, the CPU and the memory was kept low and instead the bandwidth seemed to be the bottleneck. From the test one could conclude that the proxy should be able to serve somewhere around 1920 simultaneous streams (ca 1Mbit /stream) with an average CPU load of about 80% if the bandwidth was increased to serve that amount of data.
The second test was constructed to show how the stream jitter was affected by the proxy and here we could see that the jitter actually was increased when the number of simultaneous streams through the proxy was increased. This small increase was not a big issue but this was under ideal testing environment, so this addition of jitter can in fact be disturbing on the Internet where the bandwidth might vary a lot.

8.3 Discussion

8.3.1 Results

The results from the testing of the proxy application indicated that a small average jitter was added to the client node. The added jitter from 240 simultaneous streams through the proxy (Chapter 6.5) was about 3.5 milliseconds. This small added jitter could be put in the context of how concurrent operating systems switches execution between different threads where the process schedulers typically work in 10 millisecond time quanta. With such time intervals, jitter could be introduced by the sending machine if the processing time and RTP send rate gets out of sync. So the added jitter is actually as small as the jitter inside a operating system.

8.3.2 Method

To support streaming data inside the ESB we designed and implemented a proxy application that could support one of the stream protocols in the survey (Chapter 3). This way we constructed a separate application that was build outside of the enterprise service bus. This might not be an ideal solution in terms of uniformity of the company product (iiipax). Instead, another alternative method could possibly be to look at how and if the stream could be tunneled through any of the already implemented protocols, e.g. SHS, SSEK or RIV explained in (Chaper 2). Using this approach we would have gotten the security features already implemented in the product and that we in our solution lacks. But on the other hand we would have needed to implement special encoders and decoders to tunnel the traffic.

8.3.3 Ethical aspects

Information security

This thesis investigates and implements support for a streaming protocol inside a product that are mainly used by administrators from various Swedish authorities and banks. This imposes a very high demand for information security, both integrity and confidentiality are vital components due to the nature of the information being processed. The proxy application implemented in this thesis suffers from security issues from not encrypting the streams or authenticating the nodes.
8.4. LIMITATIONS

Logging
Every stream that is being streamed through the proxy gets logged. The logging is for both non repudiation policies and business intelligence information gathering. The logging of streams could intrude in personal integrity by the ability to see inside the log what someone has streamed through the proxy.

8.4 Limitations

8.4.1 Survey
The survey was only made over three different protocols but there are more available, for example the Real Time Message Protocol (RTMP) [31] used by YouTube with Flash Player and the H.323 [32] protocol developed by the International Telecommunications Union (ITU).

8.4.2 Proxy application
While the proxy implementation is functioning and performs well under different loads, it is not yet production ready as it lacks security for authentication and integrity.

8.4.3 Testing
The tests gave a hint about how well the system performed under different loads and the results were encouraging. However, the testing was made in an ideal environment and might not reflect how the system would function over less ideal networks and on the Internet.

8.5 The future
So what could be done in the future? Here comes a list of possible features to extend and improve the final system.

- Recording could be implemented so that RTP streams could be put together and saved on disk. This would be useful in the case of recording of real-time streams such as e.g. conferences and telephone calls.

- Building in security using authentication and transport layer security mechanisms to ensure confidentiality and integrity of the streamed data. This is a crucial missing feature of the proxy solution in this thesis and must be implemented for eventual future commercial use.

- Mediation between formats could be supported. Perhaps by mediating RealNetworks’ RDT [24] protocol to RTP, the proxy could serve a wider range of media players.
• Further performance testing can be performed to ensure the robustness of the solution for example by testing a higher numbers of simultaneous stream through the proxy. Here, careful server and proxy configuration may be required to best utilize modern multi-core systems and allow the system to scale to large number of users [33].
References


List of Figures

1.1 Picture describing the scenario .......................................................... 2
1.2 Concrete example use of the service .................................................... 3

2.1 High level description of an ESB .......................................................... 4
2.2 High level figure describing session control .......................................... 6
2.3 Protocol overview of streaming scenario with RTP/RTCP .......................... 6

3.1 Protocol stack when using SIP .............................................................. 9
3.2 Example for showing the SIP infrastructure inspired by RFC 3261 [8] ....... 10
3.3 The SIP INVITE flow ............................................................................ 10
3.4 RTSP in the IP-stack ............................................................................. 13
3.5 A common RTSP client and server interaction for session setup, play and teardown .................................................................................. 14
3.6 The adaptive bitrate stream where the clients start to receive high quality pieces of a video file but suddenly change to receive a lower quality, possibly by noticing network congestion. 17

5.1 RTSP proxy system overview ................................................................. 23
5.2 Internal view over the proxy ................................................................. 24
5.3 Flowchart over the main request/response loop in the RTSP proxy. (EOF = End-of-File) ............................................................. 25
5.4 The URL to product-type table ............................................................... 26
5.5 The product-type to address table ........................................................ 26
5.6 The message log in the iipax admin web interface .................................... 27
5.7 Message diagram showing how the proxy changes the port numbers in the communication between client and server. .................. 28

6.1 The test environment topology ............................................................... 30
6.2 CPU and network utilization during different system loads ...................... 31
6.3 Memory usage during different system loads .......................................... 32
6.4 Max jitter while using and not using the proxy application ..................... 33
6.5 Average jitter while using and while not using the proxy application ........ 33
6.6 The xStream workspace inside the iipax client. . . . . . . . 34
List of Tables

5.1 The mapping between RTSP status, iipax message system status and color code. ........................................ 27

6.1 Machine specifications. .................................................. 29
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Tim Hedlund