Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

TV 2.0 – Your Individual TV Experience!

Bachelor's Thesis

Ueli Ebnöther

uelieb@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Tobias Langner
Prof. Dr. Roger Wattenhofer

July 15, 2013

Acknowledgements

First of all, I would like to thank my supervisor Tobias Langner. He was always there if I had any problems, I needed some advise or servers and he also provided me with very useful feedback during the whole thesis writing process.

Secondly I thank TPC, the Swiss television production company, for their kind help and for providing me with some test streams. Especially, I would like to thank Marcel Baur, whose feedback helped to improve the thesis remarkably and who provided me with a webspace, on which I could install my PHP services.

Then, I would like to thank Leonie Ritscher, who gave me feedback on my English and motivated me during the whole thesis.

Not to forget, I thank all the members of the Distributed Computing Group at ETH, who participated in the testings of my application. Also I thank my friends, who participated in the test over the Internet.

Last but not least, I would like to thank Professor Dr. Roger Wattenhofer for allowing me to accomplish this thesis in the Distributed Computing Group.

Abstract

The goal of this project is to enable ordinary TV users to act as real commentators for sports events and the like. To this end, we develop a solution to let interested users record their audio commentary and transmit it to the broadcasting server from where it will be redistributed in real-time to users interested in alternate audio commentaries. The end-user is able to select his preferred commentator in a web-based application that then shows the video along with the respective audio track. A crucial requirement is to ensure the synchrony of the video and the alternate audio comment.

The paper focuses on the synchronization problem, studies related work to it and introduces an own theoretical solution. The theoretical solution is then implemented in a Flash application and evaluated upon the synchronization results achieved.

With the method taken in this paper, we obtain very good synchronization results, in the area of few milliseconds. The result of this thesis allows high quality live commenting of a sports event.

Contents

Chapter 1

Introduction

1.1 Motivation

Imagine you are watching a soccer match on television. You have invited your friends, bought some chips and your favourite beer. It looks as if it is going to be a great soccer evening with your friends! But the further the match goes, the more annoyed you and your friends get because the commentator of the match is utterly bad and destroys the great soccer atmosphere. One of your friends even claims that he could be a better commentator than the person on TV.

The result of this thesis makes it possible to avoid the previous described situation and even allows your friend to be one of the commentators.

The goal of this thesis is to enable ordinary TV users to act as commentators for live sports events and the like. To achieve this, we develop a web-based application.

With this web-based application, every user who is interested in commenting the match by his own, can watch the pure (i.e. uncommented) soccer match and record his own audio commentary on the content seen. This recorded audio comment is then sent to a server, which redistributes the commentary in real-time to users interested in alternate audio commentaries.

Every user, who is interested in watching the match with another comment, can choose between different commentators from a list provided in the application. By clicking on a commentator, the chosen audio comment will play along with the corresponding visual content.

1.2 A Closer Look at the Application

This section gives a more detailed overview of the solution that we will develop to provide the users with the functionality described in the motivation section.

The solution consists of both, client and server applications. The client application will be web-based, i.e. an interested end-user can access the functionality through the web-browser. Screenshots of the implemented Flash application can be found in appendix B.

1.2.1 Streaming

Streaming is a key concept to deliver media content (such as audio and video) over the Internet. With streaming, one can play the media while transmission is still in progress. This allows the distribution of *live* media (e.g. a live sports event), which is a crucial requirement to make this application work. More details on streaming can be found in the background chapter, section 2.1.

1.2.2 The Two User Roles

As stated already in the motivation section, a user can choose between two roles:

- **Recording client:** A recording client is a user who wants to record his own audio comment. He watches the live video stream in the client application and speaks his comment into the microphone. This audio comment is recorded and will then be sent to a server. The server redistributes this audio comment to interested consuming clients.
- **Consuming client:** A consuming client is a user who does not want to record his own comment, but listens to comments from other people. A consuming client can choose a desired commentator from a list in the client application. The chosen comment then plays along with the video content.

1.2.3 The Different Streams

Figure 1.1 gives an overview of the commentary application. It shows an example with one recording end-user (for multiple recording end-users the same principles apply).

In the application we have four different data flows:

(1) We have to transmit the video data from the server to the recording clients, so that recording clients can watch the live video event.

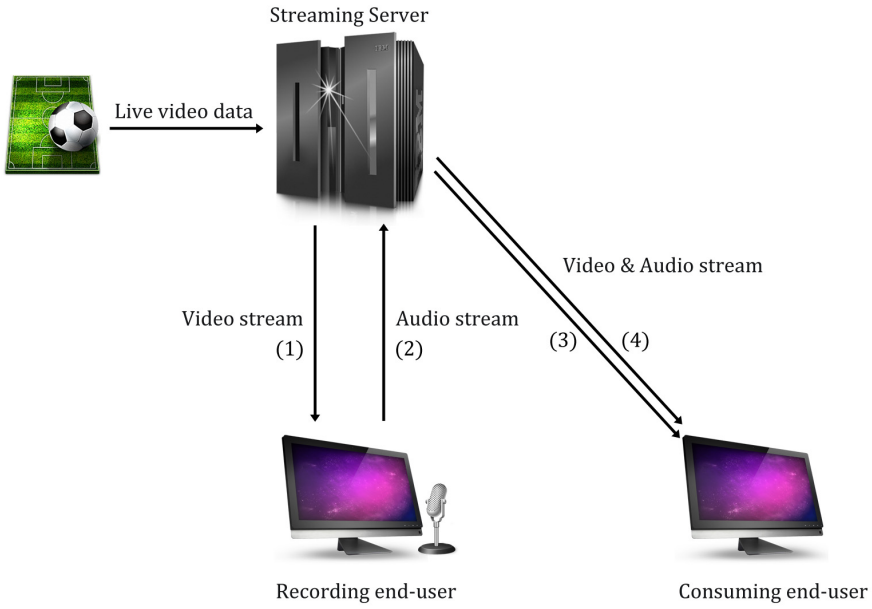

Figure 1.1: Overview of the different streams of the commentary application with the four different stream flows (in brackets).

(2) A recording client has to send the recorded audio data to the streaming server.

(3) We also have to provide the consuming clients with the video data from the server.

(4) The server has to provide consuming clients with the audio comments, that were recorded by other users.

The overall flow is as follows: The recorded live video data is passed on to a media server, which publishes the video data to connected users. A recording client plays the video and records its commentary related to the video. This recorded audio is then streamed back to the media server[1], which publishes the commentary of the user to all connected clients. A consuming client then fetches both, video and audio streams and plays them back.

[1]This server has not necessary to be the same as the video distribution server (e.g. with client side synchronization, see later).

1.3 The Main Problem to Solve

Both recording and consuming clients, see the exact same video stream. The clients (recording and consuming) watching the stream, will get the video with a certain delay (for transferring, decoding, etc.). Note, that the audio comment (related to the video content) takes a longer way (an audio comment has first to be recorded by the recording client and then redistributed by the streaming server). Therefore, it is very likely, that the consuming client will get the audio stream with a higher delay than the video stream.

The difficulty of this project is to compensate for the different delays of video and audio, i.e. to synchronize the video stream with the alternate audio commentary.

Consider this little example, on why synchronization is important: Imagine a user, who is recording, in the moment of seeing a soccer player score. Imagine your disappointment now, if you hear the corresponding audio comment "GOOAAAL!" a few seconds before you actually see the goal. Clearly, a user who watches the video with the alternate audio wants to have the scream at exactly the time he sees the goal on the screen. This requires the synchronization of the video with the alternate audio.

In this thesis we will study the problem of the synchronization in depth and provide a solution specific to the commentary application.

1.4 Main Contributions

This project made the following contributions:

- Review of available synchronization methods useful for the commentary application (chapter 3):

  We studied and evaluated available methods for the synchronization of different streams. The evaluation explained, why it is best to use which method to get a reliable commentary application.

- Theoretical solution to the synchronization problem specific to the commentary application (chapter 4):

  We solved the synchronization problem specific to this special setting of live commentators.

- Synchronization based on interval injected metadata timestamps (chapter 4):

  We developed a method to use metadata timestamps instead of packet sequence numbers or protocol timestamps included in packets.

- Implementation of the theoretical solution and evaluation (chapters 4, 5, 6):

  We implemented the commentary application in a Flash application and did an evaluation of the synchronization results of this application.

- Evaluation of the available streaming protocols (appendix A):

  We evaluated the streaming protocols available today and reasoned which one meets the application requirements best.

1.5 Chapter Overview

Chapter 2 gives some background knowledge necessary to understand the principles that will be introduced later.

Chapter 3 investigates related work on the synchronization problem, that has to be solved in the commentary application. It reviews the available methods and explains them.

Chapter 4 explains the actual theoretical solution to the synchronization problem, which is used in the commentary application. It also gives details about the implementation of the synchronization in the commentary application.

Chapter 5 gives details about the infrastructure, databases, client authentication and other features of the application.

Chapter 6 tests the implementation of the application and interprets the results achieved.

Chapter 7 concludes this thesis and states future work and ideas.

Appendix A does an evaluation of today's streaming protocols.

Appendix B contains screenshots of the developed Flash application.

Chapter 2

Background

This chapter gives some background information, that is required to understand the contents of this thesis.

2.1 Media Streaming

This section is intended to give a short introduction into streaming since it is necessary for later understanding. Because both, audio and video, have to be streamed, the term media streaming will be used in this and the following chapters.

2.1.1 The Need for Streaming

We need a way to provide the user with live video and live audio commentaries. Here, streaming comes into place. Without streaming, an end-user would have to download the whole recorded media file to play it. Using streaming, he can play received parts while transmission is still in progress. This is a crucial concept to enable live content distribution over the Internet.

2.1.2 Live Streaming Overview

In order to watch/listen to an event live over the Internet, several steps have to be followed. The most important steps, given the raw media data as input, are listed below.

Steps at sender side:

- **Encode (compress) the media:** Since raw media data is very large, we need to compress it to achieve efficient transmission over the Internet.
- **Segment the data:** The media data has to be split into small segments, which are then transmitted one-after-another to the user.

- **Packetize the segments:** To transmit these segments over the Internet, we have to encapsulate them into a packet, which contains the necessary information for delivery (e.g. IP address, checksum, ...).

Segmentation and packetization are executed by a media streaming server, which either delivers the data directly to the connected users or passes it on to a content distribution network (CDN). A CDN replicates the content among its servers and delivers the data to users connected to these servers. CDNs are used if the media content has to be delivered to a large number of people. Because then the needed availability and performance cannot be achieved by a single server. So multiple servers, which form a CDN, have to be used.

At the receiver side, we have to take the opposite steps to get the media such that it can be played:

- Get the segments out of the packets.
- Put the media segments together.
- Decode (decompress) the media data.
- Play the media data.

2.2 Why Synchronization is Needed

2.2.1 The Internet

The Internet serves as the communication medium to distribute our content to the end-users. Its growth, in terms of bandwidth and availability made it a popular choice for distributing (live) media content to people. However, streaming over the Internet is very challenging, since it does not provide any guarantees. If we deliver live content over the Internet, we encounter the following *main* problems and their consequences in regard to streaming:

- **Packets may be lost.** If we loose packets, we loose the embedded media data. This shows up as artifacts (missing frames/gaps in audio) during playback.
- **Packets are delayed.** This plays an important role in synchronization between different streams. The different delays will therefore be investigated in greater detail in the next subsection.
- **Packets may be out of order.** Good playback experience needs the media segments to be delivered in the appropriate order.

- **No guaranteed bandwidth.** Live streaming has a certain minimal bandwidth requirements. If the bandwidth drops below the requirement, the end-user encounters bad buffering during playback, rather than having a smooth media experience.

All these problems need to be handled to obtain a good media playback experience at the receiver side. Synchronization mechanisms play an important role in handling these problems.

2.2.2 A Concrete Delay Overview

The commentary application has to cope with different delays. In this subsection we look at a short overview of delays occurring at both recording and consuming clients. This will help to understand the need for synchronization in greater detail.

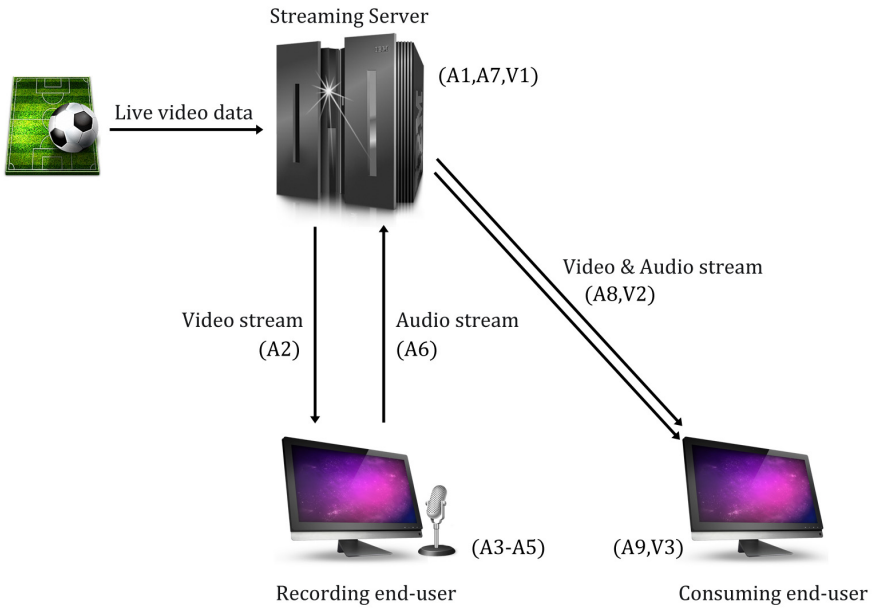

Figure 2.1: Overview of the different streams of the commentary application with the different delays (in brackets).

With a recording client the following delays are present until the audio arrives at a consuming (i.e. non-recording) client (see figure 2.1):

(A1) Delay at the video streaming server for coding, packetizing, protocol layer processing, buffering, etc. until the video is transmitted.

(A2) Transfer delay[1] of the video stream from the server to the recording client.

(A3) Delay at the recording client (decoding, buffering, etc.) until the video is displayed.

(A4) Delay of the commentator to respond for the visual content seen.

(A5) Delay until the audio comment is transferred (recording, coding, layer processing, buffering, etc.).

(A6) Delay for transferring the audio back from the recording client to the streaming server.

(A7) Delay of the streaming server for processing the incoming audio-stream.

(A8) Delay to transfer the audio to the consuming client.

(A9) Delay until the audio is played (decoding, buffering, etc.).

A consuming client gets the audio with delays (A1-A9) and the video with delays (V1-V3):

(V1) Delay at the video streaming server for coding, packetizing, protocol layer processing, buffering, etc. until the video is transmitted.

(V2) Transfer delay of the video stream from the server to the consuming client.

(V3) Delay at the consuming client (decoding, buffering, etc.) until the video is displayed.

As you can see, the audio takes a much longer way (and is therefore very likely to have a larger delay) to finally arrive at a consuming client than the video. So it is clear, that video and audio do not have the same delay, but are offset. We need a way to determine this offset and finally to compensate for the found offset to have the video content matching the audio comment.

Also notice, that these delays would be no problem, if they were all of the same length for all clients. Then the server could deliver the video with a constant delay to the consuming clients and no further synchronization mechanism would be needed. But these delays are all varying! Depending on the Internet connection (A2,A6,A8,V2), the clients playback/recording device (A3,A5,A9,V3), the client itself (A4) and the server (A1,A7,V1) all these delays can be very different (for different consuming users).

Because of the properties of the Internet, these delays are also very hard to predict.

[1] Transfer delay includes: 1. Nodal processing delay, 2. Queuing delay, 3. Transmission delay, 4. Propagation delay.

We need some mechanism to reconstruct the relation between the video and the corresponding audio comment, i.e. we need synchronization of the video and the audio, which can compensate for these varying and unpredictable delays.

Note that (A4) is not negligible. The average delay for processing visual stimulus, which is about 190ms (see [1]), has to be taken into account in further considerations.

2.2.3 Delay Requirements

An important question is, what delay between video and audio is actually detectable by the user. To investigate this, Steinmetz [2] did a user study to figure out which delays between video and audio are acceptable (i.e. not annoying to the user). The interesting results for this application are:

(1) **Lip-synchronization:** In the study, Steinmetz showed that a delay of $\pm$80ms is acceptable for audio-lip synchronization.

(2) **Pointer-synchronization:** Also, Steinmetz did an experiment, where a person explained a graphic/map by pointing on the important regions. There a delay of $-500/+750$ms is acceptable.

Conclusion: The delay between audio and video should be in the range of (2), everything else is disruptive to the user. Achieving (1) would lead to nearly perfect results. We will need these numbers later to e.g. decide if synchronization is necessary at all (i.e. for a delay under 80ms it is not really necessary to do something).

2.3 Synchronization

2.3.1 Synchronization Types

This section tries to classify our synchronization problem in order to later investigate some related work in this area. In the literature (e.g. [3],[4]) three main types of synchronization can be found:

- **Intra-stream synchronization:** Intra-stream synchronization maintains the temporal relation between the different media units (MUs)[2] of the same media stream. An example of such a temporal relation is the frame rate of a video. Intra-stream synchronization ensures that a particular stream is played out smoothly without gaps.

[2]The length of a media unit (MU) is in the range of few milliseconds.

- **Inter-stream synchronization:** Inter-stream synchronization is needed, when we have multiple separate streams with a temporal relationship. Inter-stream synchronization is responsible to reconstruct this temporal relationship.
- **Inter-destination synchronization:** Here the goal is to synchronize the playback of a media stream at different receiver locations. The media should be played at the same time at all receivers.

2.3.2 Synchronization Type in Commentary Application

In the commentary application of this thesis, we have one video stream and multiple audio streams. A consuming end-user wants to watch the video with a corresponding audio comment. As these two streams define a clear temporal relationship, but usually do not have the same delay, synchronization is needed. This synchronization corresponds to the type of inter-stream synchronization.

Chapter 3

Related Work on Inter-stream Synchronization

We need to synchronize a video stream with an audio stream to compensate for the different delays. As we saw in the last chapter, this type of synchronization corresponds to the type inter-stream synchronization. Inter-stream synchronization has been a large research area. We will look at the available methods developed by other researchers.

3.1 A Classification of the Available Methods

Ishibashi et al. [5] and Boronat et al. [6] did a comparitive survey of the available methods. Their classification of the methods can be found in table 3.1 (on page 18).

First, both classify the available methods into four groups:

(a) *Basic control techniques*, needed in most of the solutions and are essential to preserve the temporal structures.

(b) *Preventive control techniques*, needed to avoid the asynchrony (situation of out of synchrony), before it appears.

(c) *Reactive control techniques*, needed to recover from asynchrony after it has been detected.

(d) *Common control techniques*, which can be used for both prevent (prevention) and/or correct (reaction) situations of asynchrony.

These techniques are then further divided into:

(a) *Source (server) control:* The synchronization is controlled by the source.

(b) *Receiver control:* The synchronization is controlled by the receiver.

3.2 The Basic Control Idea

The basic idea is to add some information to the media units[1], such as sequence numbers or timestamps on the streams, which we want to be synchronized. The numbers, which are attached to the streams, are increasing and define a relationship between the streams. For example, if we want that a media unit of the video stream matches the media unit of the audio stream, we give both media units the same timestamp/number. The synchronization technique is then responsible, so that the numbers of the media units of both streams match during playout. If they do, (we say that) the streams are synchronized. Most techniques use some kind of buffering methods to achieve synchronization.

Figure 3.1 shows a snippet of a (past) playout situation at media unit resolution. The playout of the streams happened from left to right. You can see an example of a synchronous and an asynchronous playout of two streams.

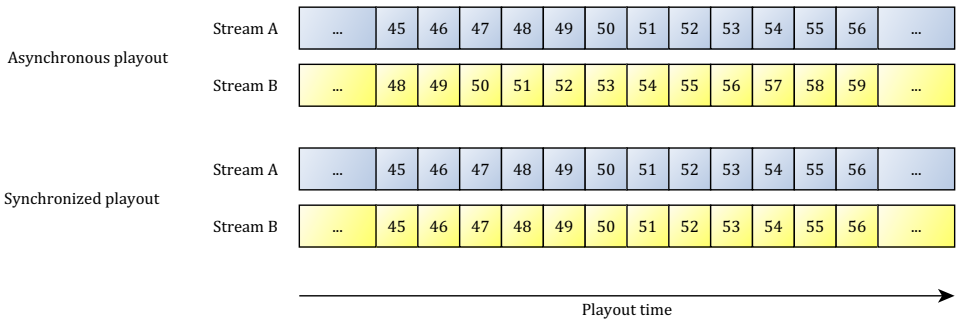

Figure 3.1: Synchronized versus asynchronous playout of two streams A and B with numbered media units.

3.3 Considerations for the Commentary Application

3.3.1 Reactive Versus Preventive Control

Since preventive control techniques estimate the delay between the streams to preventively recover from asynchrony, they are often inexact and are therefore used together with some reactive control techniques. Reactive control, on the other side, is more exact because the synchronization takes place on clearly observed asynchrony. Also, reactive control is easier to implement and more reliable, since we do not have to make any assumptions about the network. For these reasons, we will use a reactive control mechanism.

[1] Remember: A media unit is e.g. a video frame or an audio fragment of some milliseconds.

3.3.2 Reactive Source Control Versus Reactive Receiver Control

Reactive *source*[2] control needs feedback from the receiver to achieve synchronization. As this feedback has to be transferred and also the action (to avoid asynchrony) of the server is again propagated over the Internet to the receiver, this technique as well is not very exact and reliable. Reactive control at receiver-side on the other hand is more exact, more reliable and simpler to implement, since the server does not need a specialized control protocol. We will therefore use *reactive control at the receiver's side*.

3.3.3 Techniques for Reactive Control at Receivers Side

As it can be seen in table 3.1, the following techniques have been proposed by other researchers to achieve inter-stream synchronization via reactive control techniques at receiver's side:

- Reactive skips (elimination or discarding) and/or reactive pauses (repetition, insertion or stops).
- Make playout duration extensions or reductions (playout rate adjustments).
- Use of a virtual time with contractions or expansions.
- Master/slave scheme (switching or not).
- Late event discarding (event-based).
- Rollback techniques (event-based).

The next section will take a closer look at these techniques.

3.4 Review of the Techniques for Reactive Control at Receiver's Side

3.4.1 Reactive Skips (Elimination or Discarding) and/or Reactive Pauses (Repetition, Insertion or Stops)

Boronat et al. [6] state that this is a very popular idea, since it is relatively easy to implement. This technique can be explained with the help of figure 3.2.

With reactive pausing (with repetition) we stop the stream which is in advance (Stream B in the figure). Then, already played media units are repeated

[2]Source stands for the streaming server as receiver stands for a client, who receives (and plays) the stream.

to match up with the late stream (stream A). In the figure, the last three media units are repeated (49,50,51). Another option would be to only repeat the last media unit (51,51,51), which is the same as pausing the stream.

With reactive skipping/discarding, the idea is to discard already received (but not played out) media units in the late stream (stream A). In the figure, the media units (50,51,52) of stream A are discarded to match up with stream B. This requires that these media units are already in the buffer (otherwise we would always discard newly arrived packets and could not display anything).

Reactive pausing (repeating) is more feasible for the following reason: If we discard media units, we introduce gaps in the playout, so the end-user loses information, which is a big disadvantage. Also it has to be ensured, that the buffer is always full, which can be difficult, because we have a live stream and cannot get future (i.e. not yet distributed) data.

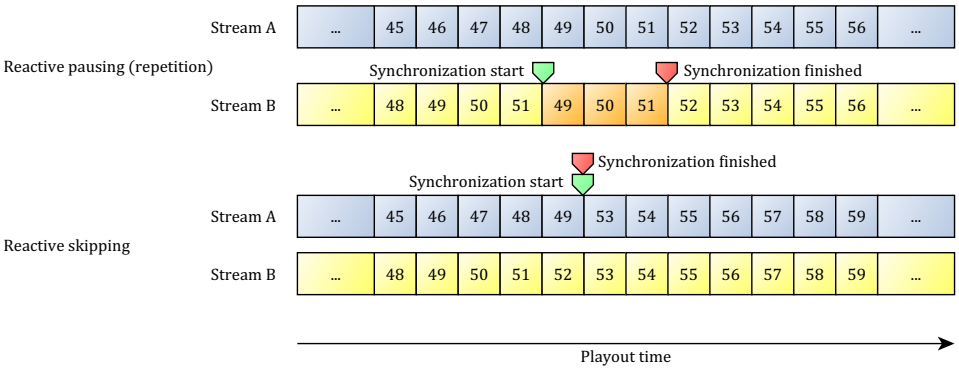

Figure 3.2: Reactive skipping and pausing (repetition).

3.4.2 Make Playout Duration Extensions or Reductions (Playout Rate Adjustments)

Another possibility to recover from asynchrony is the adjustment of the playout rate (#of media units per seconds, e.g. frames per second in a video stream). If a stream is later than the other, then we have two options: Either speed up the play out of the late stream or reduce the play out rate of the advance stream. Figure 3.3 shows the reduction of the playout rate of the advance stream (stream A) to match with stream B again.

The advantage of this technique is that we do not have to skip/pause/repeat any media units. But this technique can only be applied if the difference between the streams is not too big, since big play out rate changes would be noticed by the user, which leads to a bad playback experience.

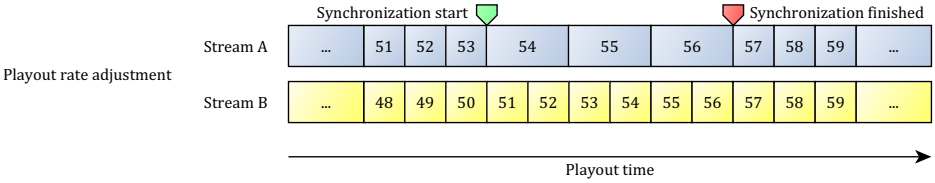

Figure 3.3: A playout duration extension (of stream A).

3.4.3 Use of a Virtual Time with Contractions or Expansions

The idea is to contract or expand a virtual time axis and then play out the media units according to this axis. This has the same *indirect* effect as increasing or decreasing the playout rate of the streams (since the playout rate depends on the virtual time). The difference is, that with virtual time contraction or expansion, all time-dependent mechanisms/methods are affected while with the other one, only the playout rate is *directly* affected. This mechanism is only an option, if you have full access to manipulate the system time.

3.4.4 Master/slave Scheme (Switching or not)

The idea is to define one stream as a master stream and the other one as the slave stream. The slave stream is then adapted to the master. Many algorithms contain switching methods, i.e. methods to switch the role of the master and the slave.

3.4.5 Event-based (Late Event Discarding and Rollback Techniques)

These techniques are mainly proposed in distributed network game applications. It does not use time, but event-based action references. This is not a suitable mechanism for the commentary application since it is more reliable to rely on time.

3.5 Additional Literature on Inter-stream Synchronization

The following papers also give some ideas on how inter-stream synchronization could be achieved.

Zhang et al. [7] as well suggest to use timestamps (they use the timestamps

present in the header of the RTP protocol packets) to synchronize different streams. They use a threshold of 200ms. If the difference between audio and video is above this threshold, they either speed up the video (if the video is later than the audio) or slow down the video playout (if the video is earlier than the audio).

The paper from Escobar et al. [8] contains some basic ideas that may be applicable for the commentary application. It proposes inter-stream synchronization under the assumption of synchronized clocks. It further states that 100-300ms for lip sync are good. They also use a buffer delay for syncing and they apply a smoothing function on delay differences to achieve good results.

Rothermel et al. [9] define one master stream and multiple slave streams, which are then adapted to the master. They use a high and a low mark where a (smoothed) buffer delay has to be in and adjust the playout (release rate) to stay between the low and the high mark.

3.6 Useful Ideas for the Commentary Application

More or less all papers propose to include some timestamps in the packets. We consider this to be the best option, since with attaching a time we can reconstruct a clear defined relationship. To conclude, the following other techniques are useful for our scenario as well:

- Use buffering techniques to delay the playback of the video or the audio.
- Reactive skipping/pausing: Drop MUs or repeat them to make the audio match the video.
- Slow/fasten playback to recover small asynchrony between video and audio.
- Smooth the delay differences.

Based on ideas in this related work, we will now introduce a theoretical solution for the inter-stream synchronization of the video with the audio comment.

Technique's purpose	**Location**	**Technique**
Basic Control	Source control	Add information useful for synchronization (timestamps, sequence numbers (identifiers) event information and/or source identifiers.)
	Receiver control	Buffering techniques
Preventive Control	Source control	Initial playout instant calculation Deadline-based transmission scheduling Interleave MDUs of different media streams in only one transport stream
	Receiver control	Preventive skips of MDUs (eliminations or discardings) and/or preventive pauses of MDUs (repetitions, insertions or stops) Change the buffering waiting time of the MDUs Enlarge or shorten the silence periods of the streams
Reactive Control	Source control	Adjust the transmission timing Decrease the number of media streams transmitted Drop low-priority MDUs
	Receiver control	Reactive skips (eliminations or discardings) and/or reactive pauses (repetitions, insertions or stops) Make playout duration extensions or reductions (playout rate adjustments) Use of a virtual time with contractions or expansions Master/slave scheme (switching or not) Late event discarding (Event-based) Rollback techniques (Event-based)
Common Control	Source control	Skip or pause MDUs in the transmission process Advance the transmission timing dynamically Adjust the input rate Media Scaling
	Receiver control	Adjust the playout rate Data interpolation

Table 3.1: A classification overview of available inter-stream synchronization methods.

Chapter 4

Solution to the Synchronization Problem

4.1 Basic Idea

Figure 4.1 again shows an overview of the commentary application, but now with timestamped segments.

Let v denote the video stream and t^v the embedded timestamps in the video stream, a the audio stream with embedded timestamps t^a and c the local clock at recording client with time t^c. The basic idea works as follows:

At the streaming server: Embed (continuously, e.g. all 10 frames) a timestamp t^v (e.g. the local server-time) in the outgoing video data.

At a recording client: Get the timestamp t^v out of the currently played video data to adjust a clock used in the recording application (if the commentator plays back video data with timestamp t^v, the recording client adjusts the local clock time t^c: $t^c = t^v$) . Because we have to compensate also for the visual reaction delay of 190ms (see 2.2.2) the client has to subtract this delay from the current timestamp. Then this current timestamp is included into the audio data. More formally: If the recording clients sees video data at clock time t^c, we set $t^a = t^c - 190ms$ and embed t^a in the currently recorded audio stream. Note that the clock c has to be stopped, if the recording client encounters buffering and has to be started again if video playback continues. Also note, that we assume, that there is no microphone recording delay.

With these timestamps, there are now two possible locations, where the synchronization can take place: At the servers side or at the consuming clients' side.

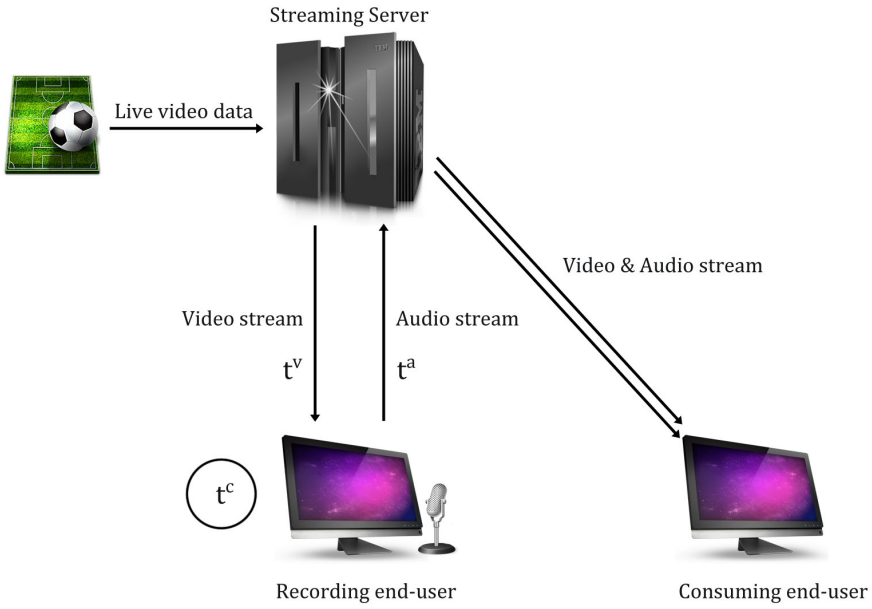

Figure 4.1: Overview of the different streams of the commentary application with timestamped segments.

4.1.1 Client-side Synchronization

For client-side synchronization, the streaming server simply distributes the audio streams with the attached timestamps to the consuming clients. The application of the consuming client then compares the timestamps embedded in the video t^v with the timestamps embedded in the audio t^a. The clients application can then compute the difference between audio and video $d = |t^v - t^a|$. Then the application can use reactive pausing/repeating or slow/fasten playback to compensate for this delay d, as proposed by related work.

4.1.2 Server-side Synchronization

At the server-side, the server computes $d = |t^v - t^a|$. Here, it is not enough to simply deliver video and audio streams at the same time (after compensated for delay d, e.g. using buffering techniques like repeating/pausing), since the transfer to the client again introduces new delays. The server would have to merge (multiplex) video and audio into a new video stream for each audio comment. So, if we would have N commentators, this would result in N different (multiplexed) video streams, all with same video content, but with another audio each. The (multiplexing) server would have to deliver all these N streams to consuming

clients at the same time (each consuming client gets one multiplexed stream with the chosen commentator). This obviously does not scale (since video has large bandwidth requirements) and would require huge infrastructure to stream the N live multiplexed video streams at the same time. Also this would require extra infrastructure to create this new N video streams at the servers side.

4.1.3 Server versus Client-side Synchronization

Server-side	**Client-side**
(−) Not scalable. (−) Costly, because more infrastructure is needed. (+) Can use any player which supports protocol and video format.	(−) Must use a specialized player at consuming clients' side[a], which synchronizes the video with the audio stream. (+) Does scale, since an audio-only stream has low bandwidth requirements. (+) Cheap.

[a]Some streaming protocols have alternate audio options. Player which support this feature then automatically synchronize the video with the alternate audio.

Table 4.1: Comparison of server-side versus client-side synchronization.

Conclusion: Since server-side multiplexing is not scalable, the decision is to choose the client-side synchronization approach. The drawback of the specialized player can be reduced, if a protocol with alternate audio support is used[1], for which players are already available.

4.2 Attaching the Timestamps

4.2.1 Type of Timestamps Used

Most protocols already have timestamps embedded in the packets (e.g. the RTMP protocol includes timestamp information in the packet header). However, in ActionScript 3 (the programming language, in which the commentary application is implemented), it is very difficult to (a) access this information and (b) include this information again. To achieve this, we would have to program an own implementation of the protocol with sockets. With sockets we would

[1]An evaluation of streaming protocols can be found in the appendix.

have full control over the packets with its bytes and we could read-out/modify the timestamp bytes for the synchronization.

A more elegant solution is to use timed metadata which is supported by most protocols. Then we can include/read out timestamp information by the natively supported methods (of ActionScript 3 and most other programming languages). The precision of this timed metadata is automatically guaranteed by the protocol (via intra-stream synchronization).

4.2.2 Attaching the Timestamps to the Video

Possible locations

In order to get the basic idea based on the timestamps work, we need to attach timestamps to the video. Remember, that the basic flow of the video is the following:

(1) Capture the live event (e.g. soccer match).

(2) Send the captured live event data to the streaming server(s).

(3) Send the data from the streaming server to the subscribed (recording or consuming) clients.

There are now two possible locations to attach timestamps to the video: After the capturing of the video data, i.e. before (2) or at the streaming server, i.e. before (3).

If the data will already be sent to different servers after (1), the timestamps have to be attached to the video immediately after (1), since all video data must have the same timestamps (in relation to same visual content) attached. If one wants to take this approach, one has to have control of the video producing site. If a TV company produces an event, this is no problem. But if they show events produced by other TV companies, it is probably not possible to get the other companies to include the desired timestamped metadata.

If we stream the captured video data to one single streaming server (and then forwarded to a CDN), we can simply attach the timestamps at this single streaming server. The solution uses this approach.

Using interval based timestamped metadata at the streaming server

The approach taken at the streaming server works as follows: Right after the video is published to the streaming server, the server starts an interval function, i.e. a function that is repeatedly executed after a specific time (e.g. every second). Every time this function is called, the server takes the current local server-time in

milliseconds (since midnight on January 1, 1970). and then generates a metadata object with key "onVideoTS" containing this timestamp. This metadata is then immediately attached to the video. The outgoing video from the streaming server then looks the same as in figure 4.2.

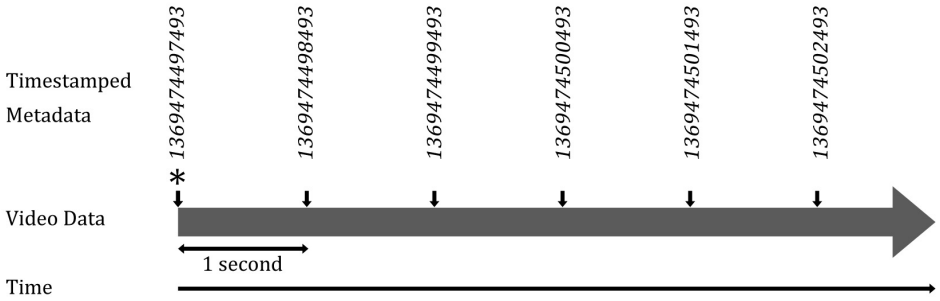

Figure 4.2: Video with timed metadata timestamps with frequency of 1 second. $*$ indicates the start of publishing.

A high interval frequency allows us faster calculation of the differences (see later) between audio and video, but introduces message[2] and work overhead at server and the clients. A low frequency requires more waiting time (due to smoothing 4.4) until synchronization can take place. So, there is a trade-off between efficiency and performance. In the evaluation (chapter 6) we took values between 200 and 1500ms, which all worked well.

To get the desired precision, the time between timestamps is later interpolated at the consuming clients (see section 4.3).

4.2.3 Attaching the Timestamps to the Audio

At a recording client, we start playing the live video stream and listen for timed metadata with key "onVideoTS". Every time, the video reaches a position with timed metadata, we do the following: Get the timestamp out of the metadata object, subtract the visual processing delay of 190ms and immediately build a new metadata object containing this timestamp with key "onAudioTS" and embed this in the currently recorded audio.

Remember that we use the assumption, that the microphone has no delay, i.e. audio is immediately transmitted without having much processing time. This ensures that video and audio stream are really synchronized if the timestamps of both streams match.

Another assumption is, that the recording client has enough bandwidth to stream to the media server (and to watch the video stream without interrupts). If

[2]Since e.g. the RTMP protocol uses separate messages to transmit the metadata.

not this would result in buffering issues which leads to permanent synchronization at a consuming client.

4.3 Get the Timestamps and Compute the Difference

Remember, as we synchronize the video with the audio at the consuming client-side, we get both audio and video as separate streams. A consuming client is now able to compute the difference between the audio and the video stream. To do this, the client plays both streams to get the timed metadata of both streams. To interpolate the time between different metadata timestamps, the client has following additional variables: A video time offset variable o_v and an audio time offset variable o_a. We now look at how to compute the difference between audio and video. The process is illustrated in figure 4.3.

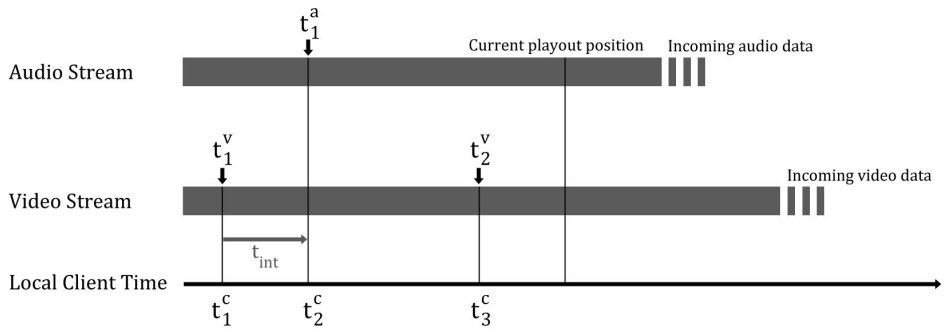

Figure 4.3: A consuming client playing both video and audio streams with embedded timestamp metadata.

Suppose we get a video timestamp with time t_1^v. The client then computes the video time offset o^v according to the local client time t_1^c:

$$o^v = t_1^v - t_1^c$$

This allows us to later interpolate the time according to the local client time. If we assume same clock drift of the server and the client and same drift of the video (assuming no buffering occurs), we get the current interpolated video timestamp t_{int}^v between events with at local client time t^c like:

$$t_{int}^v = o^v + t^c$$

So what we actually do is mapping the received timestamps to the local client time.

Suppose we get a audio timestamp with time t_1^a. We now can already compute the difference between audio and video: We take the local time t_2^c of the client

and get the interpolated time of the video:

$$t^v_{int} = o^v + t^c_2 = t^v_1 + (t^c_2 - t^c_1) = t^v_1 + t_{int}$$

Now the difference between audio and video is:

$$d = t^v_{int} - t^a_1$$

When we receive the next video timestamp t^v_2, we apply the same procedure: Interpolate the audio timestamp (with the calculated o^a variable) and get the difference. This yields a new difference (and new offset) every time a new timestamp is processed.

Another possibility would be to calculate the difference d or the offsets o^a, o^v only once. Considerations:

(1) **Calculate the difference once**, and then apply some mechanisms for synchronization based on this single difference. For example, these situations would have to be handled properly:

- The recording client encounters small buffering of the video (he gets no new video timestamp but audio recording continues):
  If the buffering of the video is very small (less than the time available in a consuming clients audio buffer) and we stop audio recording during video buffering at the recording client, the consuming clients application would not notice anything, but audio and video would be out of synchronization now. So the recording client would have to signal every time, that buffering occurred.
  With calculating the difference every time on the other hand, we would simply notice this by detecting a too high difference.
- The consuming client encounters buffering:
  Here the method with calculating the difference only once also has to calculate a new difference, because otherwise audio and video would be out of sync.

As the timestamps of the video and audio are present in the streams anyway and computing the difference takes very little time (only few lines of code to be executed), we can simply use these new differences to cope with the above problems. Also, calculating differences every time yields a permanent control over the current synchronization status.

(2) **Calculate the offset once**. This is not good practice, as the clocks of the client, the server and of the video could have different drifts. Calculating the offset every time anew helps to avoid this problem.

4.4 Smooth the Differences

For example, the differences computed[3] are the following:

968ms, 943ms, 780ms, 1020ms, 976ms, 801ms, 999ms, ...

As it can be seen, there is a small varying of the differences (in theory we would expect constant difference, if both streams are playing). The precision of these differences depend mainly on:

- The precision of these differences depend on the intra-stream synchronization of the timed metadata (with the audio or video stream).
- The precision also depends, on how exact the insertion method of the timestamps is at a recording client.
- It also depends on how exactly the events are triggered in Flash[4].
- The precision also depends on the event execution. The events of the incoming metadata are not processed concurrently[5].

The above points show, that the precision of the differences depend on many factors. We need a way to handle these fluctuations, since huge fluctuations could lead to an execution of the synchronization process, where it is actually not really needed. The method in the application uses a moving average with window size N:

- Until the (first) moment of synchronization: The method does not apply any synchronization techniques, before it has at least N differences received. After receiving the first N differences the method computes the average and decides on the required action based on this average. If e.g. no action is taken, the method waits for the next difference and then computes the average of the *last* N differences.

  For example, the first average of the first $N = 5$ differences would be:

  $$(968ms + 943ms + 780ms + 1020ms + 976ms)/5 = 937.4ms$$

  The next average would be:

  $$(943ms + 780ms + 1020ms + 976ms + 801ms)/5 = 904ms$$

[3] This data is from a real trial run.

[4] Every time Flash reaches a timed metadata object during playing, it triggers an event containing the metadata that can be processed further.

[5] In Flash events are put into an event queue and then processed one after another and not concurrently. Events which are not in the first position have to wait until the first event is processed. Flash gives no information on how long an event was waiting in the queue.

- After synchronization actions were taken: Wait for next N *new* differences, i.e. follow the procedure above.

4.5 Take a Decision Based on the Smoothed Difference

Based on the smoothed difference between audio and video we now take further steps to get synchronization. The actions are based on the times from the user study of Steinmetz [2] (see section 2.2.3). The flowchart in figure 4.4 shows the basic workflow.

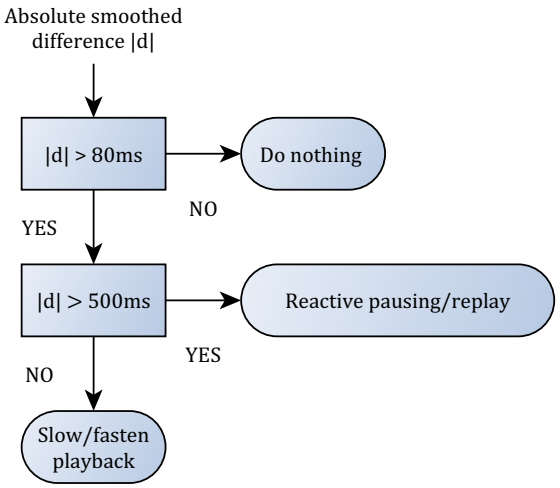

Figure 4.4: The basic actions taken by the application to handle a difference delay d.

As described in the previous section the difference is calculated as (interpolated) video time minus (interpolated) audio time:

$$d = t^v_{int} - t^a_{int}$$

There are now three cases:

- $d > 0$: If d is positive, it means that the currently played video has a higher timestamp than the currently played audio. This means that the audio is late and the video is in advance. This is the most likely case, since audio takes a longer way and is therefore very likely to have a larger delay.

- $d < 0$: If d is negative, it means that the currently played video has a lower timestamp than the currently played audio. This means that the audio is in advance and the video is late.
- $d = 0$: If d equals zero, it means that the two streams are already perfectly synchronized.

To now choose an action, we take the absolute difference of the difference $|d|$.

First $|d|$ is compared to the lip synchronization time of 80 milliseconds. If $|d|$ is equal or below 80ms we do not have to do anything, since this small difference is not remarkable for the user. If $|d|$ is higher than 80ms, we compare the delay to the pointer synchronization time of 500ms. If $|d|$ is higher than this pointer time, we use replaying of the advance stream to get a match with the delayed one, as described in the next section. If $|d|$ is less or equal to the pointer time, we want to slowly try to get a better difference with slowing or fastening the playout of the video.

4.6 Synchronization: Adjust the Playback with Reactive Pausing (Repeating)

As we argued in the related work chapter 3, it is better to use reactive pausing instead of reactive skipping/discarding, because the end-user does not miss any information.

In our solution, we use a method called "in-buffer-seeking", with which one can seek a particular time backwards in the playback-buffer (at the consuming client) during playback of the stream. This causes repetition, but allows to compensate for the computed delay $|d|$. Figure 4.5 shows an example of this technique. In this example both playback points are at the same position (this is not necessary for this technique to work properly). Now assume that the video is in advance with time $|d|$. If we now seek backwards with time $|d|$ in the video buffer and continue playback there, the video matches the audio.

We either can seek backwards in the video buffer or in the audio buffer. Where to seek depends on the sign of d: If d is positive (audio is late), we have to seek backwards by time $|d|$ in the video buffer. If d is negative (video is late), we have to seek backwards by time $|d|$ in the audio buffer.

Of course, to seek backwards in the buffers, the buffers must have enough buffered data available. If there is no data to seek backward available, we wait for buffers to fill up the required time. If buffers do not fill, the user has a low bandwidth and there is nothing we can do for him.

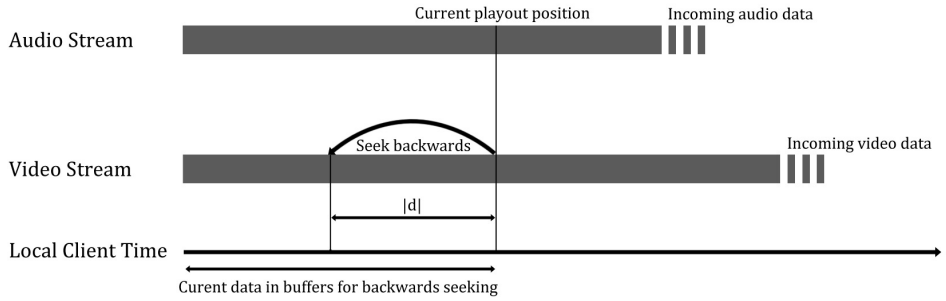

Figure 4.5: Backward seeking to compensate for difference d.

4.7 Executing Conditions

The difference computing and the synchronization procedures should not be called just when receiving a new timestamp from a playing (audio or video) stream.

The following overview gives reasons, in which situations we should call which methods.

Basically, the differences between audio and video streams should be computed every time a new timestamp in audio or video stream arrives, but there are a few extra cases (note that a (new) synchronization does not take place in these cases):

- **One of the streams is seeking:** This indicates that synchronization is taking place, so computing differences now would lead to wrong differences.
- **Synchronization took place, but we have no new video or audio timestamp:** It is important to ensure, that we only compute a difference if we have completely new audio and video timestamps (at least one of each) after synchronization. Otherwise we would compute wrong differences.
- **If a buffer of one of the streams becomes empty:** We have to reset the smoothed difference calculation (wait for new N differences), since an empty buffer which refills (and then continues playback) is very likely to get a different delay.

On the other side, synchronization should only take place if:

- **We have received at least N new differences** (see section 4.4): This is the only condition we need, since Flash procedures do not execute concurrently, a new synchronization can only take place if the old one has finished. At the end of the synchronization we reset all variables to ensure synchronization takes only place on new data.

Chapter 5

Infrastructure and Application Implementation Details

5.1 Server Setup

5.1.1 Overview

Figure 5.1 shows an overview of the infrastructure behind the commentary application.

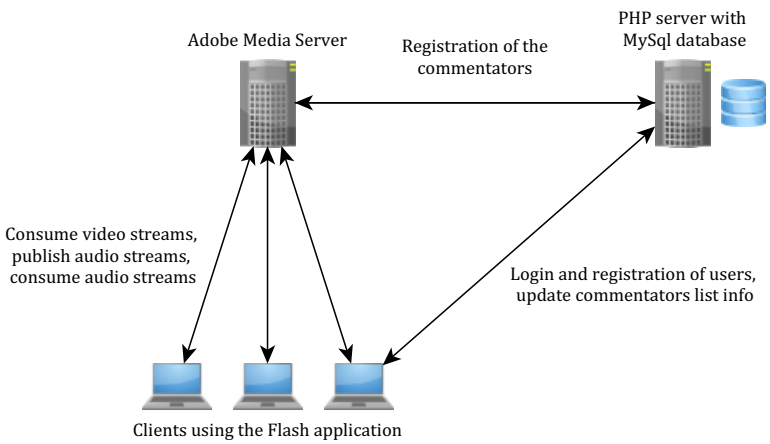

Figure 5.1: The infrastructure of the commentary application.

The infrastructure consists of two servers:

- **Adobe Media Server**: The Adobe Media Server is responsible to deliver the audio comments and the live video to consuming and recording clients and takes incoming audio comments for redistribution. Note that we can

also use two (or more) separate media servers: One for distributing the video and one (or more) to which the audio streams are published and redistributed. With many users we should pass the timestamped video to a CDN to get the desired performance.

- **PHP server with MySQL database**: The PHP server hosts services for user registration and authentication and has a service that allows the consuming clients to update their commentators list (from which a consuming client can choose the desired commentator).

5.1.2 Adobe Media Server

This project uses the Adobe Media Server Starter[1], which is available online on the Adobe website [10]. Consuming clients connect to the Adobe Media Server via the application. The server then delivers the audio and video streams to connected users and also is responsible to receive incoming audio comments and to distribute these.

Adobe Media Server allows to write server applications. This allows to do custom actions on specific "events". E.g. if a user connects, the method "application.onConnect" is invoked, which then handles the request. The server has also the ability to make HTTP calls to remote services.

5.1.3 PHP and Database Server

The PHP server hosts PHP scripts that contain services like authentication or user list updating. To invoke these services, one has to simply do an HTTP call to a specific address along with required parameters. E.g. if a user wants to register, the Flash application calls the address (POST request)

```
$host$/user_service.php?function=REGISTER
```

together with the POST variables "username" and "password".

The PHP services connect to a MySQL database hosted on the server and do modifications of this database.

The MySQL database contains the following three tables for the commentary application:

- User table: Contains username and password for authentication and a token for user validation at the Adobe Media Server (see next section).
- Streams table: Contains the currently published streams. This allows a consuming client to update the commentators list.

[1]The Starter version is for free but has limitations on the number of concurrent connections.

- Listeners table: Contains the information of which consuming client listens to which stream. This allows a nice feature which shows how many clients listen to a certain recording client.

5.2 The Application

Screenshots of the final Flash application can be found in appendix B.

5.2.1 Overview

Figure 5.2 shows the usage of the infrastructure with a recording client. Figure 5.3 shows the usage of the infrastructure with a consuming client. In both client roles, the client has first to download the application (this happens automatically when the client opens the webpage, where the Flash application is embedded). The application can be hosted on the Adobe Media Server, the PHP server or another server. Then the (consuming or recording) client begins playback of the video stream. The explanations of the other steps are given in the next sections.

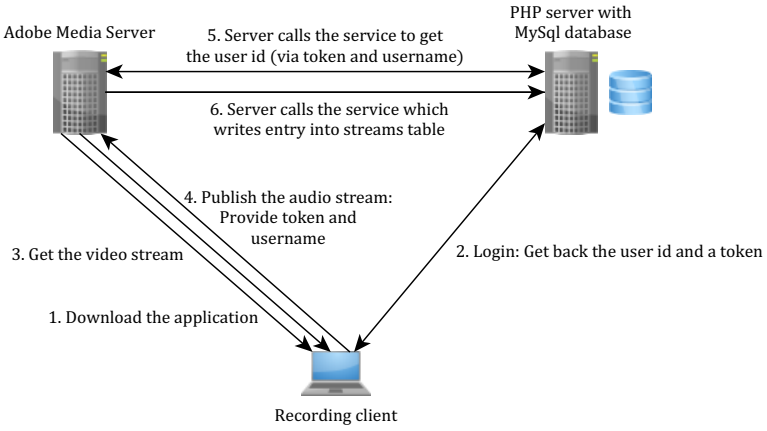

Figure 5.2: A recording client using the application.

5.2.2 Authentication of the Client

A recording client has to authenticate himself in order to publish an audio comment.

First the client registers himself with the application. This process calls a method at the remote PHP server, which writes the user information (username and password) into the database.

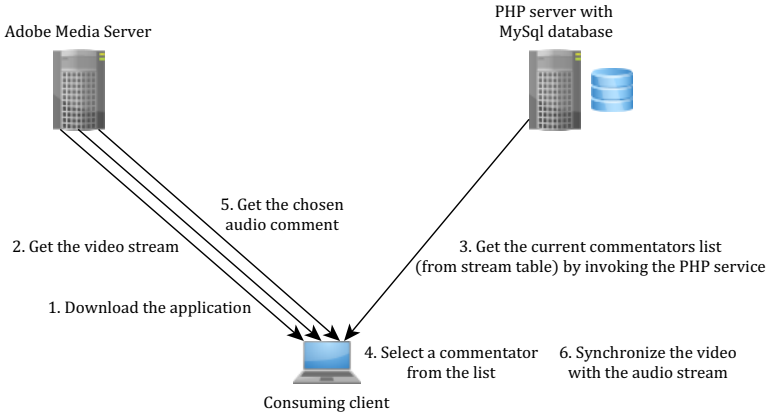

Figure 5.3: A consuming client using the application (without showing updates of the listeners).

After registration, the client can log in via a form in the application. The application validates the provided credentials via a remote PHP service and gets back the ID and a token[2] of the user.

The recording client then publishes a stream with name "audio" with appended user ID (e.g. a user with ID = 3, publishes a stream with name "audio3"). When the client connects to the Adobe Media Server, he has to provide the username together with the token received after authentication. The Adobe Media Server then looks up the user via a remote PHP service to also get the user id. This allows the Adobe Media Server to validate the stream name that should be published (otherwise, it would be possible, that a malicious user could publish audio comments in the name of another user)[3].

5.2.3 Authentication of the Server

In order to make remote PHP calls, the Adobe Media Server has to authenticate himself at the PHP server (this ensures that only the media server can do certain calls). The authentication is done via a hash (SHA 256), which is generated with a shared key[4] (between the media server and the PHP server) concatenated with command information (e.g. the stream id). The PHP server then validates this hash and only executes the corresponding function, if the hash is valid.

[2] A token consists of 32 random characters.

[3] Note that this kind of authentication is not secure against replay attacks (if used without SSL/TLS). To guard against replay attacks we would have to use an unique nonce each time or use a SSL/TLS connection (see later).

[4] The shared key is a 256bit character string. Information on key lengths [11].

5.2.4 Update the Commentators List

When a user begins publishing a new audio stream, the database at the remote PHP server has to be updated to contain the published stream. Stream database updates are completely handled at the Adobe Media Server: If a user begins to publish, the "application.onPublish" function on the Adobe Media Server is called. In this "application.onPublish" function, the media server calls the remote PHP service, which inserts a record containing the user id in the stream table of the database.

If a user disconnects or stops publishing, this is handled as follows: The "application.onUnpublish" is invoked on the Adobe Media Server. In this function, the application makes a call to another PHP service, which deletes the stream entry from the database.

There is an important reason, why the media server does the calls to update the commentators list (and not the client itself): If a user closes the application, the media server can detect this by a timeout and do a remote call to delete the entry in the streams table. This ensures that the streams table only contains entries from commentators which are really publishing.

The consuming clients application can grab the current commentators via a remote PHP call and update the list (from which the end-user can choose a commentator) with the information received.

5.2.5 Update the Listeners Table

We want to provide a consuming client with the information of how many clients listen to which audio commentator. The inserts and deletes into/from the listeners table are done by the Adobe Media Server. This has two advantages:

- If a consuming client closes the application, the media server detects a timeout of the connection and can do a delete from the database.
- The Adobe Media Server only writes an entry into the database if the client is really connected.

This ensures that we have only an entry in the database if the client is really connected.

To update the listeners table, a consuming client calls a function on the Adobe Media Server, which does a remote PHP call to insert an entry with the client id and the commentator id. If a client disconnects, this entry is deleted.

To provide consuming clients with the number of listeners for each commentator, we simply count the number of entries with the according commentator id.

The listeners table also allows us to show the recording clients, how many listeners they have.

5.3 Security

As both the client application and the media server do HTTP calls to a remote PHP server, an attacker could intercept the messages (along with the POST variables) and replay/modify these to modify the database. To secure against this, we can use a SSL/TLS connection for both between the media server and the PHP server and between the application and the PHP server. This prevents not only from leakage of the request (for example, this prevents that an attacker can get the username and the password when doing a login) but also prevents against replay attacks, since SSL/TLS does generate a MAC[5] which is unique each time.

As security is not the main topic of this thesis, the Flash implementation contains only provisional security features.

[5]Message Authentication Code.

Chapter 6

Evaluation

6.1 An Intuitive Test

In this test, we streamed a video of a clock and commented every thirty seconds with a peep. The results of this test, of course, depended on the reaction time of the specific user (and the microphone delay, which is assumed to be 0).

The test participants reported that the peep indeed matches the clock if the numbers of the measured synchronization offset (difference between audio and video after synchronization) in the application are low. Of course this is not an exact test, but it justifies that the method actually really does what it is supposed to do.

In the next sections, we are going to show offsets measured by the application.

6.2 Application Measured Synchronization Results

To evaluate the synchronization, we used a setup of two media server instances. One for distributing the video and one for distributing the audio.

6.2.1 Evaluation Setup and Details

How we Measured the Offsets/Differences

The results that are shown in the next sections are the measured *(non-smoothed)* differences from the application (computed by the application as described in section 4.3). Of course the precision of these differences depend on the precision of the intra-stream synchronization of the timed metadata and the precision of the Flash event processing (as explained in 4.4).

Interval Frequency and Smoothing Parameter

For the first evaluation, we took an interval frequency (see 4.2.2) of 1500ms and smoothed (see 4.4) over the 8 newest differences . With this setting, we get a video timestamp every 1500ms as well as a audio timestamp every 1500ms. So synchronization can take place after less than 10 seconds (if we have enough data to seek backward in the buffers).

For the second evaluation, we took an interval frequency of 200ms and smoothed also over the 8 newest differences. With this setting, we get a video timestamp every 200ms as well as a audio timestamp every 200ms. So synchronization can take place after less than 1 second (if we have enough data to seek backward in the buffers). In the third setting, we used 300ms.

Slowing and Fastening of the Video

The technique of slowing and fastening of the video (introduced to slowly recover from little asynchrony) was not implemented in the application (see 6.3.2) and therefore no further synchronization happens when the offset (after synchronization) is between the pointer synchronization time (magenta lines in plots).

The used Test Stream

To test the whole system, we had a video of the FIFA World Cup Final 2010 (Netherlands vs. Spain). Then we streamed it, as if it was live, with FFmpeg[1] [12]. Used settings for the stream:

- Format: flv
- Resolution: 640x362pixels
- Bitrate: 606 kbit/s
- Codec: H.264 Main Concept
- Framerate: 24fps

6.2.2 Two Recording Clients, Three Consuming Clients, 1500ms, Local Network

Setting: 1500ms interval, wait for 8 differences. Tested in the internal Ethernet ETH network.

[1]Description from the FFmpeg project website ([12]): "FFmpeg is the leading multimedia framework, able to decode, encode, transcode, mux, demux, stream, filter and play pretty much anything that humans and machines have created."

This small test (of length 2 minutes and 30 seconds) was done with only two recording clients and three consuming clients (each on a separate machine). The synchronization numbers in milliseconds reached can be found in the following diagrams (figures 6.1, 6.2, 6.3). The blue line shows the offset (measured in the application) between audio and video. The horizontal green lines indicate $\pm 80ms$, the horizontal magenta ones $\pm 500ms$. The vertical dashed black lines indicate switching to other audio comments.

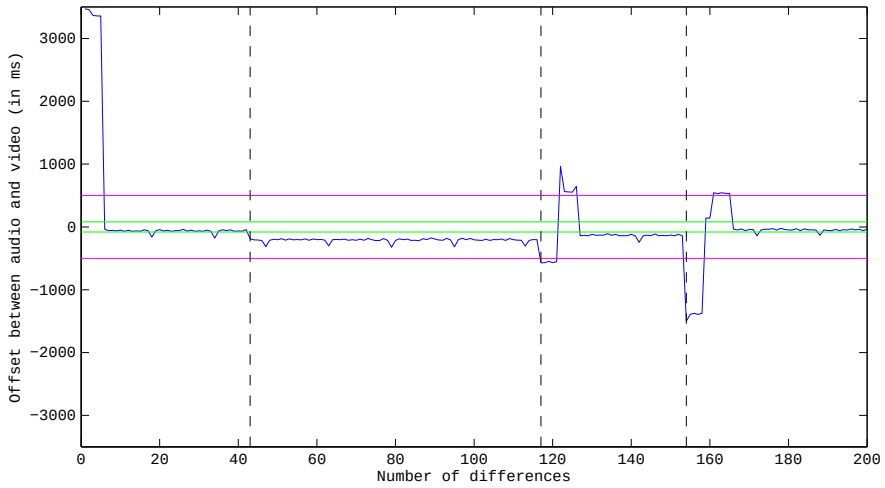

Figure 6.1: Test #1: Synchronization results of consuming client number 1.

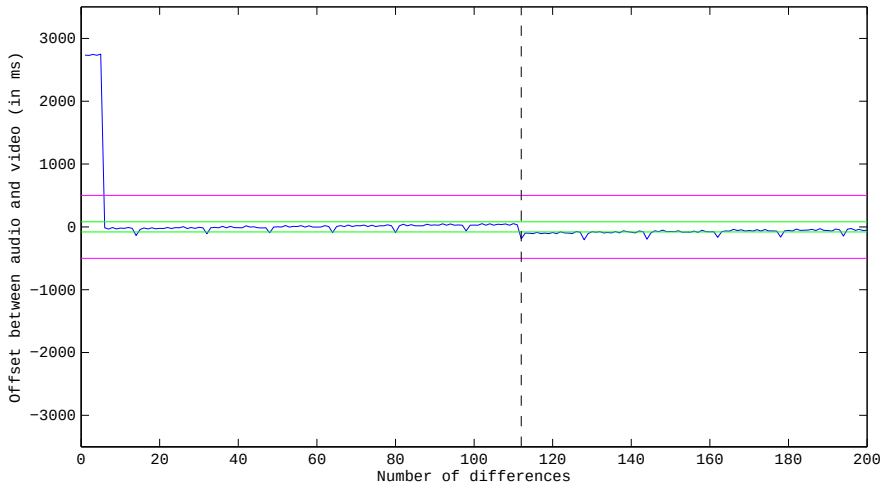

Figure 6.2: Test #1: Synchronization results of consuming client number 2.

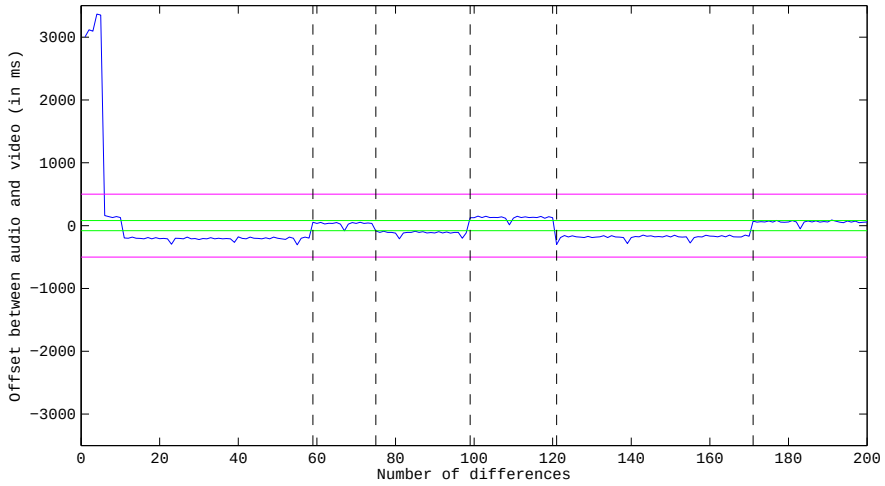

Figure 6.3: Test #1: Synchronization results of consuming client number 3.

Interpretation of the results

- At the beginning, there is an offset of more than 3 seconds in all consuming clients. This initial offset is immediately corrected in all cases.
- Seeking compensates the initial offset correctly, but because the seeking method in Flash takes also some milliseconds to execute, we get a new (but very low) offset.
- When synchronization took place, the offset stayed more or less constant. There are fluctuations due to the precision factors described in 4.4.
- After synchronization, the offset stayed between pointer time ($\pm 500ms$). Often, the offset is in the range of lip synchronization ($\pm 80ms$), which is a very good result.
- The gaps, after switching to another audio comment, are introduced because the consuming client has to wait until the audio buffer fills with new audio and until enough differences (due to smoothing of the differences, 4.4) are present to apply the seeking method.
- Different audio comments often have the same delay, so after switching we do not need to seek to a new position (since we corrected the delay by seeking the video, of which the playback continues during an audio channel change. E.g. figure 6.3 needed no new seeking at all.). This is due to the fact, that the delay is mostly due to the encoding at the media server. The transfer delay of the audio comments play only a minor role in the test

setup. It would have become more important, if we would have used a CDN, because there we introduce additional delays (because we have to transfer the media contents to different servers).

6.2.3 Two Recording Clients, Three Consuming Clients, 200ms, Local Network

Setting: 200ms interval, wait for 8 differences. Tested in the local home network.

Since the interval is 200ms, the total length of this test (figure 6.4) is 70s. We show only results of consuming client number 1.

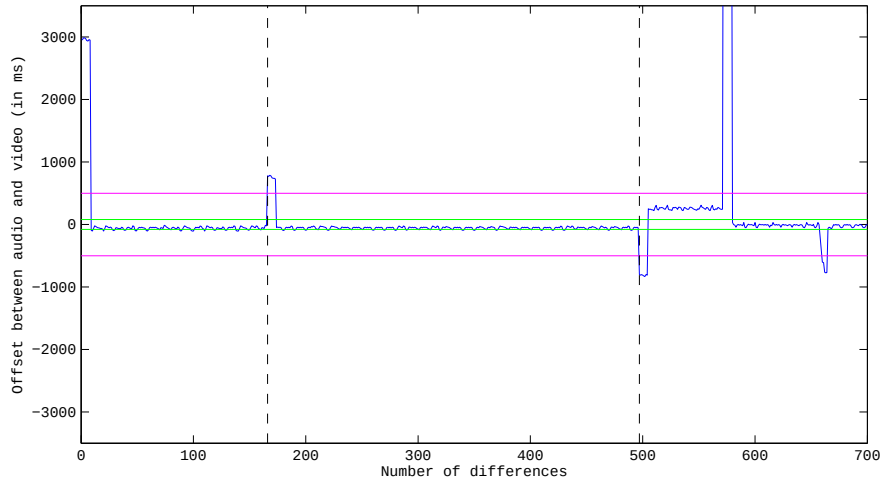

Figure 6.4: Test #2: Synchronization results of consuming client number 1.

Interpretation of the results (additional to previous interpretations)

- Due to the high frequency of 200ms, synchronization could take place only after one second, after buffers filled up.
- The high peak around difference number 600 is due to the low wireless bandwidth in the used home network. The buffer of the audio was emptied at the consuming client and caused this high difference. After buffer filled up again, synchronization correctly compensated for the new difference.

6.2.4 Three Recording Clients, Four Consuming Clients, 300ms, Internet

Setting: 300ms interval, wait for 8 differences. Tested over the Internet.

Note that consuming clients had the following initial offsets, which were corrected: 2382ms, 2392ms, 2418ms, 2618ms.

The results can be found in figures 6.5, 6.6, 6.7, 6.8.

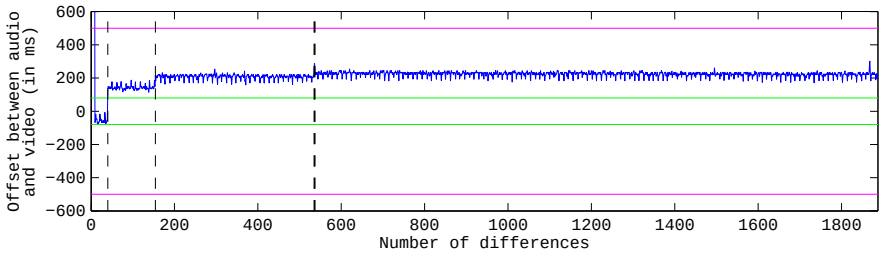

Figure 6.5: Test #3: Synchronization results of consuming client number 1.

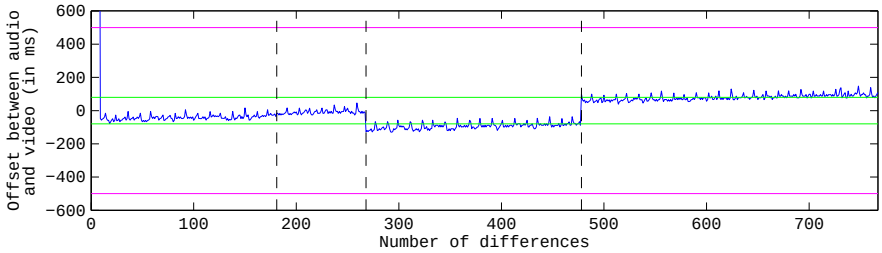

Figure 6.6: Test #3: Synchronization results of consuming client number 2.

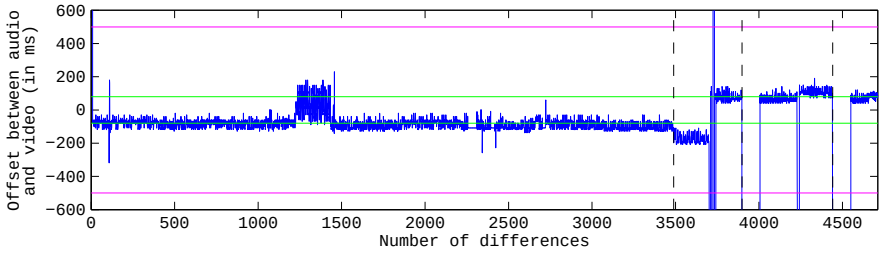

Figure 6.7: Test #3: Synchronization results of consuming client number 3.

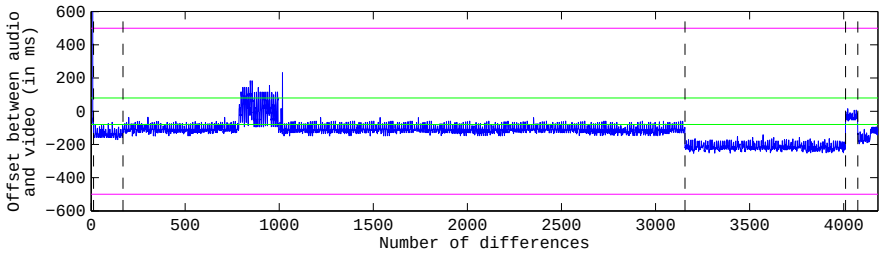

Figure 6.8: Test #3: Synchronization results of consuming client number 4.

Interpretation of the results (additional to previous interpretations)

- The results are in the pointer synchronization range, which is a good result.
- After switching or synchronization, the offset stays constant. This offset could be corrected further by slowing or fastening.
- Client 3 has some huge peaks, due to buffering issues.
- Client 3 and 4 have some low peak around 1300ms, 700ms respectively. This is due to temporary bandwidth problems at the recording client. (Client 1&2 started later, after this happening).

6.3 Problems Encountered (and Solutions)

6.3.1 Seeking is not Exact

Seeking in Flash is not as exact as it could be. The remaining differences are not very high and could be compensated by slowing or fastening of the video.

6.3.2 Problems with Slowing and Fastening of Video

Unfortunately, ActionScript 3 does not have a native feature to manipulate the frame rate of the video. There are some work arounds proposed by the Internet community, but all of these are not reliable enough and result in strange behavior. Therefore the slowing and fastening of the video is completely omitted in my source code (but indicated with a comment where it should take place with which action).

6.3.3 Microphone Delay

We did the assumption, that the microphone has no recording delay. On some devices, there can be a remarkable delay (over a second), which destroys the

synchrony of audio and video. One solution to this problem would be to do a calibration process, where the delay is determined with help of the user and this delay is then taken into account during timestamp insertion into the audio.

6.3.4 Adobe Media Server Starter Version

The Adobe Media Server Starter version only supports 10 connections at a time. So the system only could be tested with maximum 4 consuming and recording clients (1 connection is needed for the in-going video stream and each client needs 2 connections (one for video and one for audio)). With two separate servers (one for the video stream and one for the audio streams), we could make tests with maximum 9 clients.

6.3.5 HTTP Based Streaming Protocol

In the appendix A, I motivated to use an HTTP based protocol, since it has many advantages. For the playback of HTTP streams, there exists an ActionScript library (Open Source Media Framework). Unfortunately, I did not manage to read/insert timestamps with this library, this is why I used the RTMP protocol and the native AS3 functions.

6.3.6 Buffering Problems

As seen in the evaluation, if the recording client or the consuming clients buffer get empty, we have to wait of course until the buffer is full again. Then we need to again synchronize. We cannot do anything against buffering, we have to assume, that the clients have enough bandwidth for the application to work as otherwise the whole approach is doomed to fail.

Chapter 7

Conclusion

We developed an application that allows people to comment a live sports event. Other people can listen to recorded live audio comments by choosing a commentator in the list provided by the application.

The thesis studied related-work to inter-stream synchronization, because it is a crucial requirement, that the audio comment stream is synchronized with the video stream at consuming clients.

We introduced an own solution to the synchronization problem in this specific setting, which inserts timestamps in the streams and uses these to calculate an offset between the audio comment stream and the video stream. We used timed metadata timestamps, which we interpolated at the consuming client to get the needed precision to calculate the offset. We used backwards seeking in the buffers by this offset in the streams to make the audio synchronous with the video.

As the evaluation results show, the commentary application can achieve really good synchronization results. We achieved that the offset between audio and video (after synchronization) lies between the pointer synchronization time, as desired and motivated. Often the offset even stays in lip-synchronization time, which is almost perfect.

7.1 Future Ideas/Work

7.1.1 Microphone Delay

As already mentioned in section 6.3, the solution only works, if we assume zero microphone delay. A future idea is to implement a microphone calibration test to get rid of this assumption.

7.1.2 Test on a CDN

The test results shown in chapter 6 are captured from a small test setting. What has still to be investigated is, how good the solution performs in a large setup (over a CDN).

7.1.3 Switching to Other Audio Comments

As seen in the evaluation, switching to other audio comments can cause reseeking. A very nice thing would be, if switching to other comments would be seamlessly, without new syncing. This would require the delay of the different audios at the server's side. For example, the streaming server could delay all audio streams, that they all have the same delay. This would be difficult, when using over a CDN and would therefore have to be investigated in detail.

7.1.4 Different Video Streams

Another nice feature, would be, that a consuming user could not only switch to different audio streams, but also to different video streams. For example, we could stream different views of a soccer match, from which the user could choose.

7.1.5 Bandwidth Measuring

We should deny users with low bandwidth the recording of a comment, because it leads to a bad experience at the consuming clients side. For denying, a possibility would be to run a bandwidth test before the client can begin publishing and only grant access if the bandwidth is high enough.

Bibliography

[1] Kosinski, R.J.: A literature review on reaction time. Clemson University **10** (2008)

[2] Steinmetz, R.: Human perception of jitter and media synchronization. Selected Areas in Communications, IEEE Journal on **14**(1) (1996) 61–72

[3] Ud Din, S., Bulterman, D.: Synchronization techniques in distributed multimedia presentation. In: MMEDIA 2012, The Fourth International Conferences on Advances in Multimedia. (2012) 1–9

[4] Wu, D., Hou, Y.T., Zhu, W., Zhang, Y.Q., Peha, J.M.: Streaming video over the internet: approaches and directions. Circuits and Systems for Video Technology, IEEE Transactions on **11**(3) (2001) 282–300

[5] Ishibashi, Y., Tasaka, S.: A comparative survey of synchronization algorithms for continuous media in network environments. In: Local Computer Networks, 2000. LCN 2000. Proceedings. 25th Annual IEEE Conference on, IEEE (2000) 337–348

[6] Boronat, F., Lloret, J., García, M.: Multimedia group and inter-stream synchronization techniques: A comparative study. Information Systems **34**(1) (2009) 108–131

[7] Zhang, J., Li, Y., Wei, Y.: Using timestamp to realize audio-video synchronization in real-time streaming media transmission. In: Audio, Language and Image Processing, 2008. ICALIP 2008. International Conference on, IEEE (2008) 1073–1076

[8] Escobar, J., Partridge, C., Deutsch, D.: Flow synchronization protocol. Networking, IEEE/ACM Transactions on **2**(2) (1994) 111–121

[9] Rothermel, K., Helbig, T.: An adaptive stream synchronization protocol. In: Network and Operating Systems Support for Digital Audio and Video, Springer (1995) 176–189

[10] : Adobe media server Available online at http://www.adobe.com/products/adobe-media-server-family.html; visited on June 25th 2013.

[11] : Bluekrypt cryptographic key length recommendation Available online at http://www.keylength.com/en/; visited on June 28th 2013.

[12] : Ffmpeg Available online at http://www.ffmpeg.org/; visited on July 3rd 2013.

[13] Parmar, H., Thornburgh, M.: Adobe real time messaging protocol (rtmp) (2012) Available online at http://www.adobe.com/devnet/rtmp.html; visited on May 2th 2013.

[14] Schulzrinne, H.: Real time streaming protocol (rtsp). (1998)

[15] Apple: Http live streaming. Available online at https://developer.apple.com/resources/http-streaming/; visited on May 4th 2013.

[16] Adobe: Adobe high dynamic stream protocol (hds) Available online at http://www.adobe.com/ch_de/products/hds-dynamic-streaming.html; visited on May 4th 2013.

[17] Microsoft: Microsoft smooth streamin (mss) Available online at http://www.iis.net/downloads/microsoft/smooth-streaming; visited on May 4th 2013.

[18] Sodagar, I.: The mpeg-dash standard for multimedia streaming over the internet. MultiMedia, IEEE **18**(4) (2011) 62–67

[19] : Web rtc Available online at http://www.webrtc.org; visited on May 4th 2013.

Appendix A

Appendix Chapter

A.1 Streaming Protocols

In this chapter, we will give a short overview of the available streaming protocols and evaluate the most used protocols for live media streaming which are available today. The evaluation compares these protocols and focuses on those which meet the protocol requirements the best. The protocols are divided into two sections: Protocols for streaming video data from the server to a client and protocols to stream recorded audio from a recording user to the streaming server.

A.1.1 Protocol Requirements

The desirable properties of the protocols for the commentary application are:

- The protocol should allow **good quality experience** during playback.
- To achieve inter stream synchronization, we need the protocol to be able to carry some **timed metadata** (such as a timestamp) in the stream, to later reconstruct a temporal relationship between audio and video.
- It is also better, to choose a protocol, that has a **wide support** by as many clients as possible.
- It is desirable, that there are **web-based** players available which support the protocol.

A.1.2 Alternate Audio Support

Alternate audio support by the protocol means, that we can define one video streams along with multiple alternative audio streams (e.g. in a manifest file). Alternate audio support has the advantage that all players, which support this

protocol, can be used. These players allow switching between the alternate audio tracks. Therefore it is not necessary to program a new media player for synchronization at a consuming client.

Alternate audio would have been an elegant solution, but unfortunately, this requires low level access to the timestamp information in the packets. Since this is difficult with ActionScript 3 (would require own socket protocol implementation), we use another route with timestamped metadata (see solution chapter 4) and implement our own player to have full access over the synchronization process. This also has the advantage, that we can provide the user with extra features (which we would not have with standard players).

A.1.3 The Different Transport Protocols

Two main transport protocols for media streaming (from server to client and vice-versa) are used nowadays: The User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP). Table A.1 gives a short overview of the advantages and disadvantages (in terms of streaming) of the protocols. This will also help in choosing a streaming protocol later, since they are all built up on top of either UDP or TCP.

UDP	**TCP**
(−) Lost packets are not retransmitted.	(−) Retransmission mechanism introduces additional delay.
(+) Faster in most cases[a], since no reliability mechanisms.	(+) Retransmission of lost packets.
	(+) Ordering of packets.
	(+) Congestion control.
	(+) Flow control.

[a] This depends on the net infrastructure. Congestion can make it slower than TCP.

Table A.1: Comparison of UDP and TCP

The non-reliability properties (in terms of loss and ordering) of UDP can introduce artifacts and bad quality experience during media replay. UDP is the preferred choice for time-critical media streaming, because it is faster in many situations, which is a crucial requirement for applications such as Skype™. With TCP we have to use a larger buffer time, since retransmission should not be noticed by a pause in the video. For live media it is feasible to use TCP, since it does not matter if the user gets the content with (e.g. 10 seconds) more delay than with UDP.

Today's trend goes towards the usage of the Hypertext Transfer Protocol (HTTP) (which builds on top of TCP) streaming protocols. The next section will try to explain this tendency.

A.1.4 Streaming from the Server to the Client

This section analyses the protocols for streaming from the server to the client (i.e. for streaming video from the server to recording clients and for streaming the video/audio from the server to the consuming clients).

A.1.5 The Advance of HTTP for Media Streaming

A main category for streaming the video from the server to the clients are HTTP-based protocols. HTTP-based protocols work as following: The client downloads a manifest or playlist file, in which the available media segments are listed. The client then uses those references to request and download the media segments via HTTP.

The main reason for the release of new streaming protocols based on HTTP by some major companies, such as Apple, Microsoft and Adobe, are the following advantages:

- HTTP builds on top of TCP and provides quality playback experience and encounters no firewall problems.
- HTTP is a widely-implemented, well known protocol which is supported by many devices.
- Normal HTTP web servers can be used to stream the media. Servers need no special protocol implementation.
- As HTTP is a pull-based protocol, the user requests the next segments. Therefore the server has to maintain very little user state information.
- Already widely adapted HTTP caching infrastructure can be used.
- Support for adaptive streaming available (see next subsection).

Because of these advantages, it is preferable to use an HTTP based protocol, even if we encounter a bigger latency since is is based on TCP.

Adaptive or dynamic streaming

Adaptive or dynamic streaming[1] is a technology, that allows the user to switch dynamically between streams of different bitrate (and therefore quality). The server has to encode the stream in different qualities. If the player encounters a bandwidth drop, it can switch dynamically to a lower bitrate. Vice-versa he can also switch to a higher quality if the bandwidth allows it. This allows the user to choose the best playback experience (i.e. without interrupts/buffering) for his bandwidth conditions.

Overview of media streaming protocols

This section gives an overview of the most popular media streaming protocols. After a short introduction to every protocol, this section summarizes advantages and disadvantages in table A.2. For further information on these protocols take a look at the references.

This section concentrates on streaming video and audio from the server to the client. Protocols for streaming the audio to the server are analysed in the next section.

Adobe Real Time Messaging Protocol (RTMP)

RTMP is a protocol widely used nowadays. For example, online Web-TV companies like Zattoo or Wilmaa are using this technology along with a flash based webplayer to deliver their content. RTMP is based on TCP and can use an HTTP tunneling mode to avoid firewall problem (this is called RTMPT). Timed metadata is supported and can easily be injected at a recording client and read out at the consuming client using a Flash application. The protocol specification is available at [13].

Real Time Streaming Protocol (RTSP)

RTSP is a relatively old protocol (published 1998). For the actual data stream RTSP uses the RTP protocol and RTCP protocol for QoS management. RTSP can use either UDP or TCP as transport protocol. There is no web-based player like Flash to play an RTSP stream[2]. See [14] for the protocol specification.

[1] *Dynamic streaming* is the terminology used by Adobe with conjunction with the RTMP protocol.

[2] You can only watch these streams in a browser with a browser plugin like VLC Player.

Apple HTTP Live Streaming (Apple HLS)

As the name suggests, this is an adaptive HTTP based protocol. It is natively supported by iOS and OS X. Since there are also Flash plugins available which can play such streams, this is a very interesting possibility. It supports adaptive bitrates. Metadata can be embedded via ID3 tags, which are embedded in the MPEG-2 transport stream. For a live stream, the playlist (of type .m3u8) containing the available segments have to be continuously downloaded to get the newest available segments.

Adobe High Dynamic Streaming (Adobe HDS)

This also is an adaptive HTTP-based protocol. The segments are specified in a .f4m file, which has to be continuously downloaded by the client for a live stream. Although this is a protocol from Adobe, it is not natively supported by Flash. The Open Source Media Framework (OSMF) library would have to be used to develop an application, which can play this kind of stream. Methods for timed metadata are similar to those available for RTMP. See [16] for more information on Adobe HDS.

Microsoft Smooth Streaming (MSS)

Contrary to the previous two protocols, the manifest file for this protocol has to be downloaded only once. The next segments for a live stream can then be computed autonomously by the client. This protocol also has timed metadata support. When it came out, it was only supported by the Microsoft Silverlight browser plugin. Now there is a Flash library available to enable Flash support. MSS is also supported by Windows Phones with version greater or equal to 7. Information can be found at [17].

MPEG Dynamic Adaptive Streaming over HTTP (MPEG DASH)

MPEG DASH may be the future protocol for streaming. It meets all the application requirements, but documentation and support are still very limited. So this may be a choice for the future, but not for now. [18] gives an overview of DASH.

Overview

Property	RTMP	RTSP	Apple HLS	Adobe HDS	MSS	MPEG DASH
Support						
HTML 5	No	No	No[a]	No	No	No
Flash (native)	Yes	No	No[b]	No[b]	No[b]	No[b]
iOS (native)	No[c]	No[c]	Yes	No	No	No[c]
Android (native)	No[c]	Yes	Yes[d]	No[e]	No[e]	No[e]
Technology						
HTTP based	(Yes)[f]	No	Yes	Yes	Yes	Yes
Adaptive	Yes[g]	No	Yes	Yes	Yes	Yes
Requirements						
Timed metadata	Yes	Yes	Yes	Yes	Yes	Yes

[a] Only in Safari (on iOS and OS X), not in other browsers.
[b] Flash plugins/libraries available.
[c] Libraries/solutions for apps available.
[d] In Android Versions 3. Since 4.2 support is discontinued.
[e] Only android versions with Flash Player.
[f] A HTTP tunneling mode is available.
[g] This is called dynamic streaming.

Table A.2: Comparison of streaming protocols

Conclusion

As HTTP streaming has many advantages (see section A.1.5), we only consider HTTP-based protocols. As MPEG DASH support is still very limited, this is no option nowadays. Therefore, the recommendation is to use any of the following three protocols for streaming to video and the audio from server to the client:

- Adobe RTMP protocol (with HTTP tunneling).
- Apple HTTP Live Streaming (HLS).
- Adobe High Dynamic Streaming (HDS).
- Microsoft Smooth Streaming (MSS).

Which protocol to choose depends basically on the programming language used for the application. Since we use ActionScript 3 and the RTMP protocol is the only natively supported protocol, we implement the application with this protocol (to do a proof of concept).

A.1.6 Audio Streaming to the Server

This section gives an overview on which audio streaming protocols could be used to stream the recorded audio from the recording client to the streaming server.

For streaming the audio from a recording client to the server we cannot use a HTTP protocol since HTTP is pull-based. Instead we have to use a push-based protocol to push the recorded audio data from the client to the server, like RTMP or RTSP.

To record and stream audio to the server with a web-based application, there are only two option available: Flash or HTML5 (WebRTC [19]). Since WebRTC is very new and only supported in the Google Chrome web-browser, we use a Flash based solution. For a Flash based application, it is best to use the RTMP protocol, since it is the only natively Flash supported protocol for streaming to a server.

Appendix B

Application Screenshots

B.1 Start screen

The start screen allows to choose if you want to comment a video or if you want to be a consuming client (figure B.1).

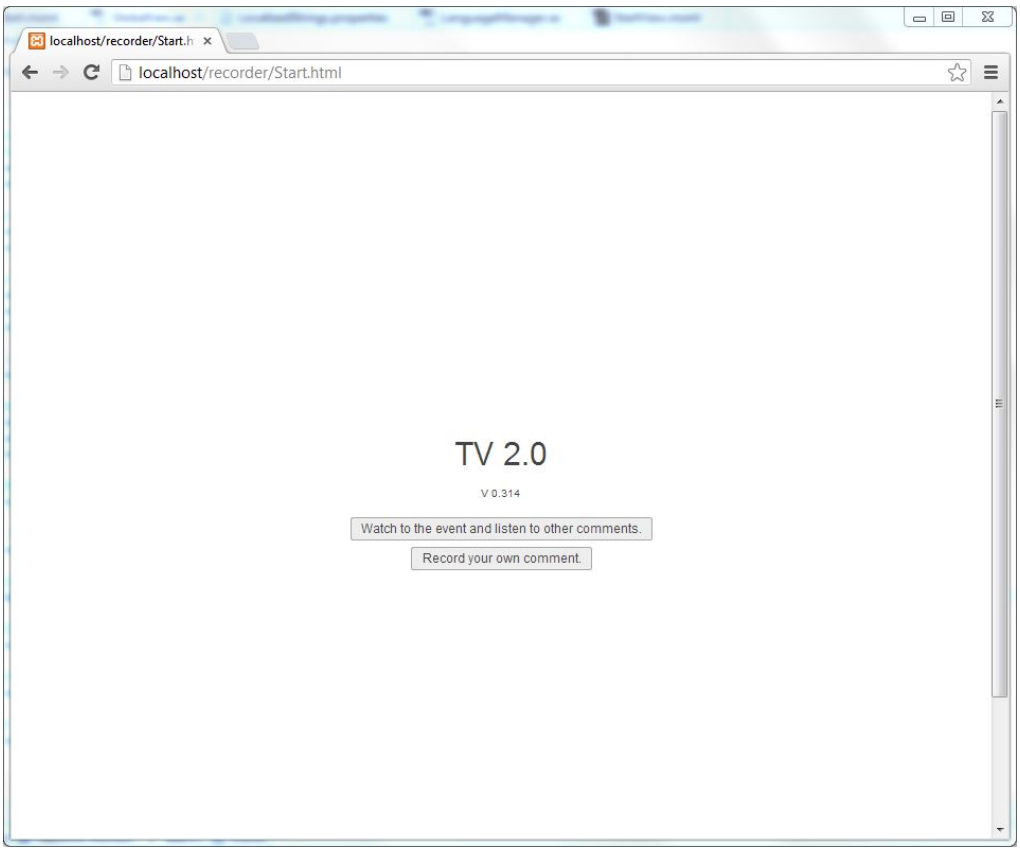

Figure B.1: Start screen of the Flash application.

B.2 Recording user

B.2.1 Login

The recording user first has to register and has to log in in order to publish an audio comment (figure B.2).

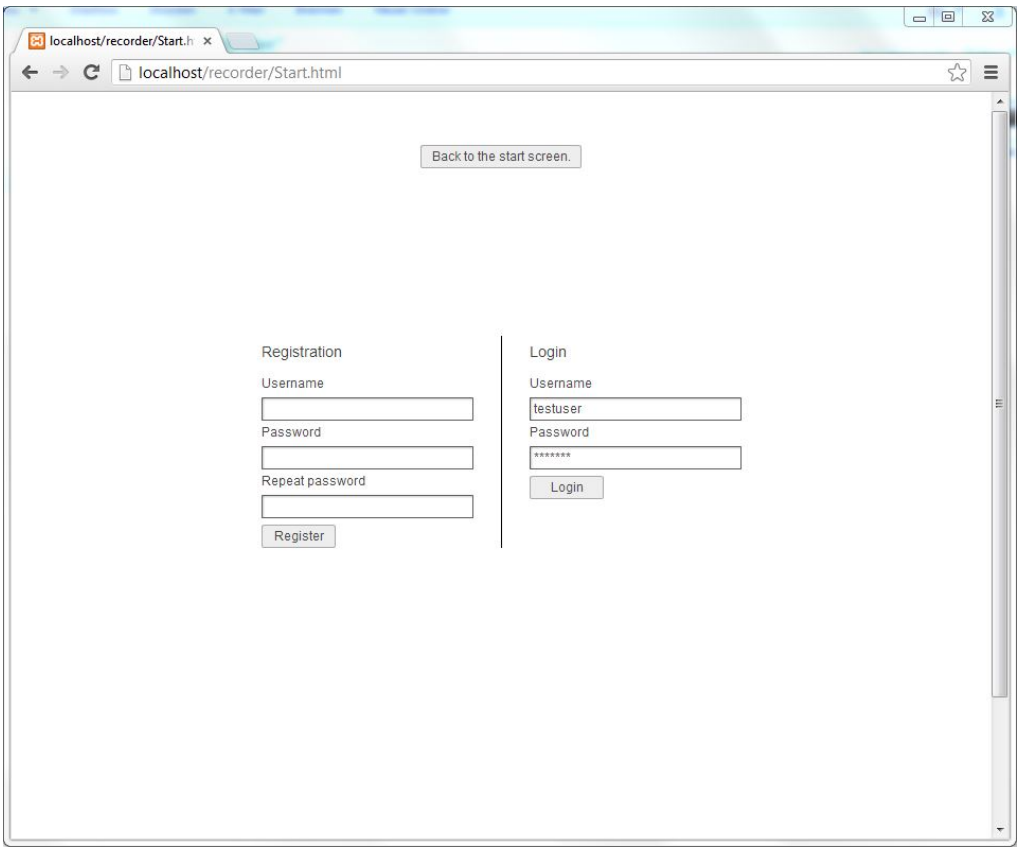

Figure B.2: Login screen of the Flash application.

B.2.2 Audio recording

If logged in, the recording user can record his own audio comment to the video. He sees the video and can adjust the volume of it. Also he sees a green bar, indicating the volume of the microphone and he can adjust the microphone volume. The recording client also sees how many listeners he currently has. See figure B.3.

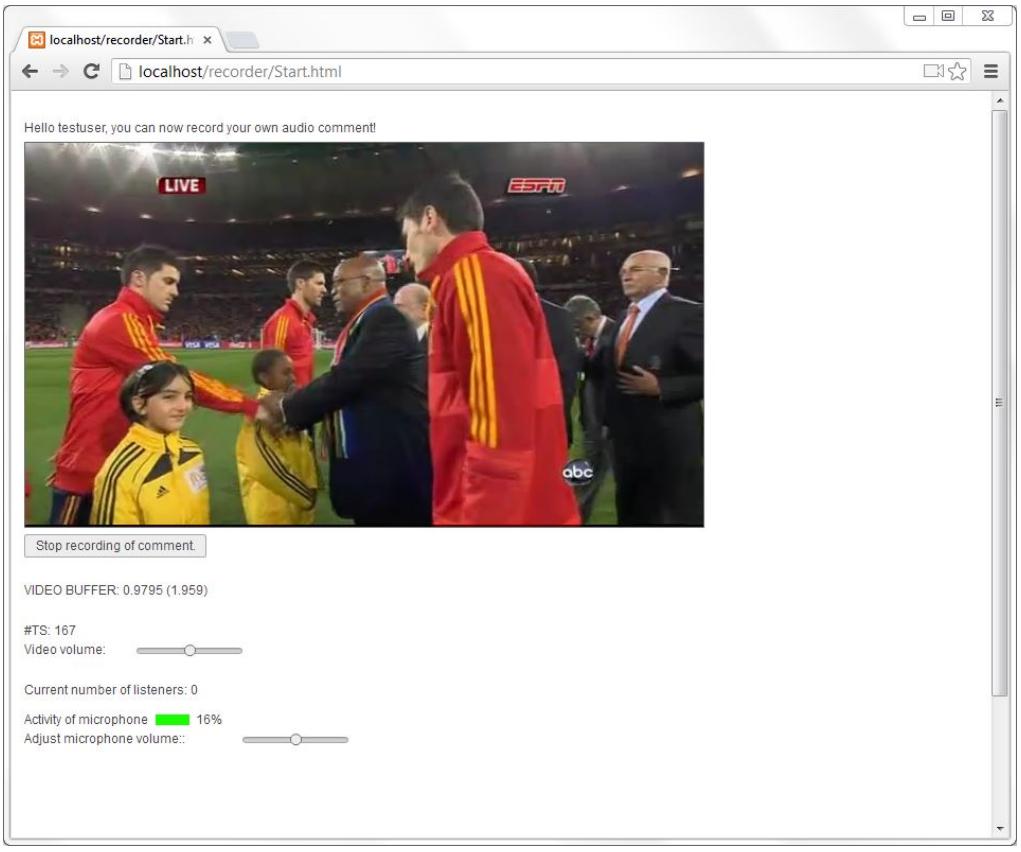

Figure B.3: Comment recording interface of the Flash application.

B.3 Consuming client

A consuming client can choose a commentator from the list on the left. The video plays along with the audio comment. At the bottom, there is an indication if the synchronization and network are OK (additional with the current measured offset in milliseconds). The consuming client can adjust the volume of the video and the audio. See figure B.4.

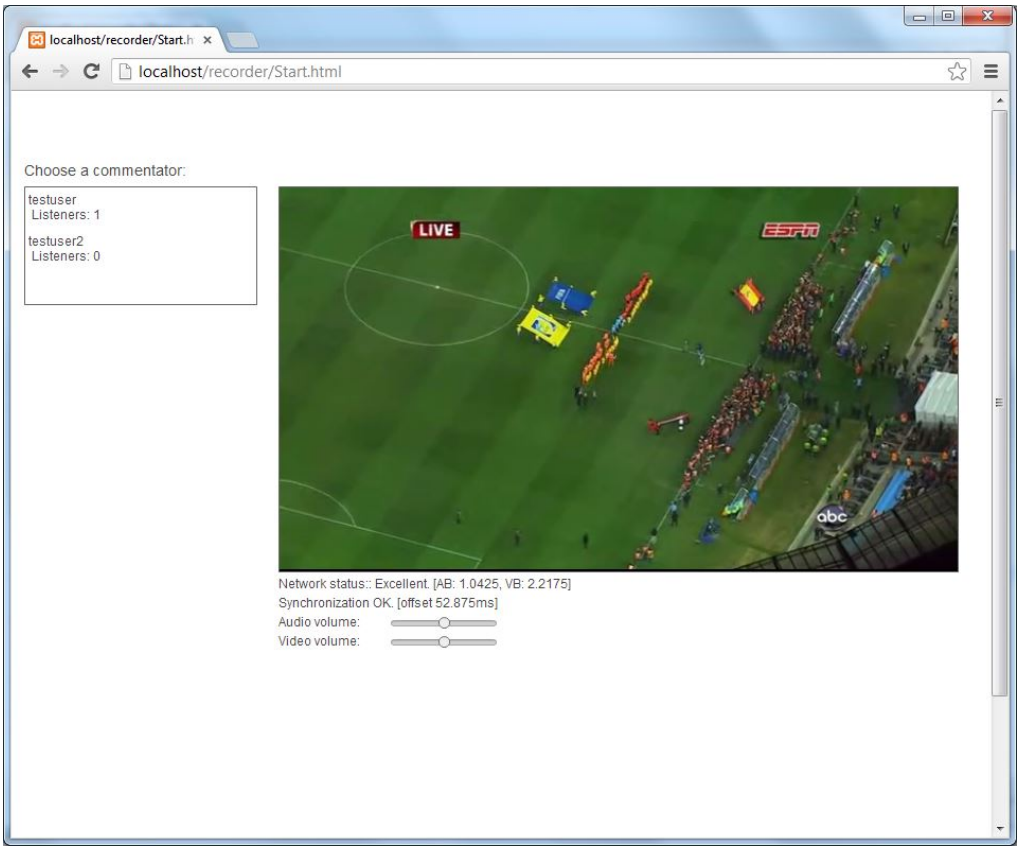

Figure B.4: Consuming client screen.