IEEE International Conference on Multimedia Computing & Systems, June 17-23, 1996, in Hiroshima, Japan, p. 372-381

Intra- and Inter-Stream Synchronisation for Stored Multimedia Streams

Ernst Biersack, Werner Geyer, Christoph Bernhardt
Institut Eurécom
2229 Route des Crêtes, 06904 Sophia Antipolis, FRANCE
{erbi,geyer,bernhard}@eurecom.fr

Abstract

Multimedia streams such as audio and video impose tight temporal constraints due to their continuous nature. Often, different multimedia streams must be presented in a synchronized way. We introduce a scheme for the continuous and synchronous delivery of distributed stored *multimedia streams across a communications network. We propose a protocol for the synchronized playback, compute the buffer requirement, and describe the experimental results of our implementation. The scheme is very general and does not require bounded jitter or synchronized clocks and is able to cope with clock drifts and server drop outs.*

1. Introduction

1.1 Motivation

Advances in communication technology lead to new applications in the domain of multimedia. Emerging high-speed, fiber-optic networks make it feasible to provide multimedia services such as Video On-Demand, Tele-Shopping or Distance Learning. These applications typically integrate different types of media such as audio, video, text or images. Customers of such a service retrieve the digitally stored media from a **video server** [1] for playback.

1.2 Multimedia Synchronization

Multimedia refers to the integration of different types of data streams including both **continuous media** streams (audio and video) and **discrete media** streams (text, data, images). Between the information units of these streams a certain temporal relationship exists. Multimedia systems must maintain this relationship when storing, transmitting and presenting the data. Commonly, the process of maintaining the temporal order of one or several media streams is called **multimedia synchronization** [4].

Synchronization can be distinguished on different levels of abstraction. *Event-based* synchronization assures a proper orchestration of the presentation of distributed multimedia objects. A multimedia object may be, for instance, a news cast consisting of several subobjects like audio and video. On a lower level *continuous synchronization* or *stream synchronization*, respectively, copes with the problem of synchronizing the playout of data streams [13]. The classical example of stream synchronization is the *lip-synchronized* presentation of audio and video [5].

Continuous media are characterized by a well-defined temporal relationship between subsequent data units. Information is only conveyed when media quanta are presented continuously in time. For video/audio the temporal relationship is dictated by the sampling rate. The problem of maintaining continuity within a single stream is referred to as **intra-stream** synchronization. Moreover, there exist temporal relationships between media-units of related streams, for instance, an audio and video stream. The preservation of these temporal constraints is called **inter-stream** synchronization. To solve the problem of stream synchronization, we have to regard both issues which are tightly coupled.

One can distinguish between **life synchronization** for life media streams and **synthetic synchronization** for stored media streams [15]. In the former case, the capturing and playback must be performed almost at the same time, while in the latter case, samples are recorded, stored and played back at a later point of time. For life synchronization, e.g. in teleconferencing, the tolerable end-to-end delay is in the order of a few hundred milliseconds only. Consequently, the size of the elastic buffer must be kept small, trading-off requirements for jitter compensation against low delay for interactive applications. Synthetic synchronization of recorded media stream is easier to achieve than life synchronization: higher end-to-end delays are tolerable, and the fact that sources can be influenced proves to be very advantageous as will be shown later. It is, for instance, possible to adjust playback speed or to schedule the start-up times of streams as needed. However, as resources are limited, it is desirable for both kinds of synchronization to keep the required buffers as small as possible. [9]

1.3 Context of the Synchronization Problem

The synchronization problem addressed in this paper is motivated by our work on scalable video servers. We have designed and implemented a video server, called **Server Array**, consisting of *n* **server nodes**. A video is distributed over all server nodes using a technique called **sub-frame striping:** Each video frame f_i is partitioned into *n* equal size parts $c_{i,j}$, called **sub-frames**, that are stored on the *n* different servers. If $F_i = \{c_{i,1}, \ldots, c_{i,n}\}$ denotes the set of sub-frames for f_i, then: $\bigcup_{j=1\ldots n} c_{i,j} = f_i$

The server array with the synchronization mechanisms presented in this paper has been successfully implemented in our video server prototype [1]. During playback, each server node is continuously transmitting its sub-frames to the client. The transfer is scheduled such that all striping blocks that are part of the same frame are completely received by the client at the deadline of the corresponding frame. The client reassembles the frame by combining the sub-frames from all server nodes.

Another example for inter-stream synchronization of stored multimedia streams is given by Cen et al. [2]. They describe a distributed MPEG player with the audio server and the video server being at different locations in the Internet environment.

2. Synchronization Scheme

2.1 Overview

We propose a synchronization scheme for the delivery of *stored media* that achieves both, suitable intra- and inter-stream synchronization. The scheme is *receiver-based* and does not assume global clocks. Resynchronization is done by skipping/pausing, and furthermore, we apply the concept of a buffer level control. To initiate the playback of a stream in a synchronized manner we introduce a novel **start-up protocol**. Our protocol has been influenced by the ideas of Ishibashi et al. [8] who achieve inter-stream synchronization by providing intra-stream synchronization for each stream involved and by Santoso [14] who provides conditions for a smooth playout. For *re*-synchronization, we adopt a scheme similar to the one described by Koehler et al. and Rothermel et al. [9], [13].

We derive our synchronization scheme by step-wise refinement: First we develop a solution for the case of zero jitter and then relax this assumption requiring bounded jitter only. Finally, we cover synchronization problems not introduced by jitter. In each step we derive the buffer requirements and playout deadlines to assure inter- and intra-stream synchronization. We present three models

Model 1 solves the problem of *different, but fixed delays* on the network connections for each substream. We propose a synchronization protocol that compensates for these delays by computing well-defined starting times for each server. The protocol allows to initiate the synchronized playback of a media stream that is composed of several substreams.

Model 2 takes into account the *jitter* experienced by media-units travelling from the source to the destination. Jitter is assumed to be bounded. To smoothen out jitter, elastic buffers are required. Our scheme guarantees a smooth playback of the stream and has very low buffer requirements. Model 2 covers intra-stream synchronization as well as inter-stream synchronization.

Model 3 solves the problems of *clock drift*, *changing network conditions* and *server drop outs* by employing a buffer level control with a feedback loop to the servers so as to regain synchronization in the case of disturbances. Again, buffer requirements are regarded with respect to the results of models 1 and 2. The behavior of a filtering function is examined. Filters are necessary to identify whether a problem is of long-term or short-term effect. The tuning of some parameters is discussed.

For the proposed synchronization scheme, we assume that a client **D** is receiving sub-streams from different servers[1]. Client and servers are interconnected via a network (see figure 1).

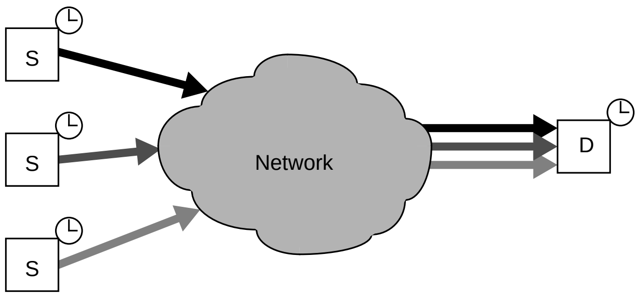

Figure 1. Distributed architecture for the synchronization scheme

Each of the servers denoted by *S* delivers an independent **substream** of **media-units** (sometimes also referred to as **frames**). The production rate is driven by the server clock. Arriving media-units are buffered in FIFO queues at the destination **D**. The playout of the entire **stream**, composed of all the substreams, is driven by the destination's clock.

[1]It is also possible that a *single* server sends *multiple* substreams to a client. Our model is more general and covers this case too.

2.2 Sources of Asynchrony

Several sources of asynchrony exist in the configuration described in the previous section. These are: Different delays, network jitter, end-system jitter, clock drift, alteration of the average delay, and server drop outs.

2.3 Assumptions

Our synchronization mechanism uses **time stamps**. Each media-unit[2] scheduled by a server is stamped with the current local time to enable the client to calculate statistics, such as for the roundtrip delay, jitter, or inter-arrival times. Moreover, we assume that each media-unit carries a **sequence number** for determining media-unit order. We could use for our purposes a protocol such as RTP, which is currently designed by the IETF, and which provides fields for both, time stamp and sequence number.

In contrast to other approaches, buffer requirements or fill levels are always stated in terms of media-units or time, instead of the amount of allocated memory. This seems reasonable because media-unit sizes vary due to encoding algorithms like JPEG or MPEG [9]. However, notice that a mapping of media-units to the allocation of bytes must be carried out for implementation purposes. Taking the largest media-unit of a stream as an estimate wastes a lot of memory, especially when using MPEG compression. Sophisticated solutions of mapping are subject of future work. In the following, we will use the term **buffer slot** to denote the buffer space for one media-unit.

Since processing time, e.g. for protocol actions does not concern the actual synchronization problem, we will neglect. Finally, we assume that control messages are reliably transferred.

Table 1 shows the parameters that we used to describe our model. A set of media-units that needs to be played out at the same time is referred to as **synchronization group**.

We assume that media-units are distributed in a *round robin fashion* across the involved server nodes. Hence, we can identify the storage location of a media-unit by its media-unit number[3], i.e.

$$\text{Server } S_{i \bmod n} \text{ stores media-unit } i. \qquad (1)$$

This leads to the following formulation of the **synchronization problem:**
The client must playout the media-units of all subsets I_j, with $j \bmod n = 0$, at the *same time.*

[2]We will also use the abbreviation **mu** for media-unit.

[3]This implies that each substream will send media-units at the *same rate*. An extension of the scheme to different media-unit rate, each one being the integer multiple of a base rate is straight forward.

Symbol	Description	Unit
n	number of server nodes in the server array	
N	number of media-units of a stream	
i, j, υ	media-unit index ($i, j, \upsilon \in \{0, \ldots, N-1\}$)	
k	server index ($k \in \{0, \ldots, n-1\}$)	
I_j	index set of n subsequent media-units starting with media-unit j	
S_k	denotes server node k providing substream k	
D	denotes the destination or client node	
s_i	initial sending time of media-unit i in server time	[sec]
s_i^c	synchronized sending time of media-unit i in server time	[sec]
a_i	arrival time of media-unit i in client time	[sec]
d_i	roundtrip delay[a] for media-unit i measured at client site	[sec]
d^{max}	maximum roundtrip delay	[sec]
t_{start}	starting time of the synchronization protocol	[sec]
t_{ref}	reference time for the start-up calculation	[sec]
t_0	earliest possible playout time of the first media-unit	[sec]
t_i	expected arrival of the media-unit i at the client site	[sec]
δ_{ij}	arrival time difference between media-unit i and j	[sec]

Table 1. Model parameters

a The roundtrip delay comprises the delay for a control message that requests a media-unit and the delay for delivering the media-unit

2.4 Model 1: Start-Up Synchronization

2.4.1 Introduction

Under the assumption of constant delay and zero jitter, we solve the synchronization problem by assuring that the first n media-units, which constitute a synchronization group, arrive at the *same time* at the client. We therefore *need*

$$t_i = t_0 \qquad \forall i \in I_0 \qquad (2)$$

The major problem addressed by model 1 is the compensation for different delays due to the independence of the different substreams. For instance, the geographical distance from server to client may be different for each server. Thus, starting transmission of media-units in a synchronized order would lead to different arrival times at the client with the result of asynchrony. Usually, this is compensated by delaying media-units at the client [5]. Depending on the location of the sources, large buffers may be required.

In order to avoid buffering to achieve the equalization of different delays, we take advantage of the fact that stored media offers more flexibility: The idea is to initiate playout at the servers such that media-units arrive at the sink in a synchronous manner. This is performed by shifting the starting times of the servers on the time axis in correlation to the network delay of their connection to the client. The proposed start-up protocol consists of two phases.

- In the first phase, called **evaluation phase**, roundtrip delays for each substream are calculated, while
- In the second phase, called **synchronization phase**, the starting time for each server is calculated and transmitted back to the servers.

The model is based on the assumption of a *constant end-to-end delay* without any jitter. For the moment we do not consider changing network conditions, server dropouts, and clock drift. In such a scenario, synchronization needs to be done once at the beginning and is maintained afterwards automatically.

We need to introduce some more notation to express interdependencies between the parameters of the model. We then give a description of the start-up protocol flow and prove its correctness. We close the section with an example of the protocol.

The starting time t_{start} of the protocol equals the beginning of the first phase. Without loss of generality let $t_{start} = 0$.

To begin with, we regard the first *n* media-units of a stream given by I_0 that are distributed across the n servers.

The second phase of the protocol begins at time t_{ref}, determined by the last of the first *n* media-units that arrives: $t_{ref} = max\{a_i | i \in I_0\}$

The difference $\delta_{ij} = a_i - a_j \quad \forall i, j$ between the arrival times of arbitrary media-units *i* and *j* is needed to calculate the starting times of the servers.

2.4.2 Start-Up Protocol

The synchronization protocol for starting playback at the server sites is launched after all involved parties are ready for playback and consists of an evaluation phase and synchronization phase. During start-up, the client sends two different kinds of control messages to the servers:

- *Eval_Request(i)*: Client *D* requests media-unit *i* from server S_i, $\forall i \in I_0$.
- *Sync_Request(i,* s_i^c*)*: Client *D* transmits the starting time s_i^c to server S_i.

(a) Evaluation Phase

- At local time t_{start}, client *D* sends an *Eval_Request(i)* to servers S_i, $\forall i \in I_0$.
- Server S_i receives the *Eval_Request(i)* at local time s_i, $\forall i \in I_0$.
- Server S_i sends media-unit *i* time-stamped with s_i immediately back to client *D*, $\forall i \in I_0$.
- At local time a_i, client *D* receives media-unit *i* from Server S_i, $\forall i \in I_0$.
- At local time t_{ref}, client *D* has received the last media-unit.
  The roundtrip delays $d_i = a_i - t_{start}$, $\forall i \in I_0$ and the maximum round trip time $d^{max} = max\{d_i | i \in I_0\}$, are computed.

(b) Synchronization Phase

- At local time t_{ref}, client *D* computes
  the earliest playout time $t_0 = max\{t_{ref} + d_i \mid i \in I_0\}$,
  the index υ that determines t_0 as $\upsilon = \{j \in I_0 | t_{ref} + d_j = t_0\}$, and
  the delay differences as $\delta_{\upsilon i} = a_\upsilon - a_i$, $\forall i \in I_0$
- With these results the starting time of Server S_i is calculated in server time
  $s_i^c = s_i + d^{max} + \delta_{\upsilon i}$, $\forall i \in I_0$.
- Client *D* sends a *Sync_Request(i,* s_i^c*)* to server S_i, $\forall i \in I_0$.
- At local time $s_i + d_i + (t_{ref} - a_i)$, server S_i receives the *Sync_Request(i,* s_i^c*)*, $\forall i \in I_0$.
- At local time s_i^c, server S_i starts scheduling of the substream by sending media-unit *i*, $\forall i \in I_0$.
- At local time t_i, client *D* receives media-unit *i*, $\forall i \in I_0$.

At any time, only one synchronization group of *n* media-units must be buffered at the client; after the complete reception the media-units are played out immediately.

2.5 Model 2: Intra- and Inter-Stream Synchronization

2.5.1 Introduction

Model 1 shows how to cope with different but constant delays for each substream. However, synchronization is performed under the assumption that jitter does not exist. Model 2 loosens this assumption and takes into account *end-system jitter* and *network jitter.* We consider the cumulative jitter and assume the jitter to be bounded.

Due to jitter, media-units will not arrive in a synchronized manner although they have been sent in a timely manner. The temporal relationship within a single substream is destroyed and time gaps between arriving media-units vary according to the occurred jitter. Thus, an isochronous playback cannot be achieved if arriving media-units of a substream would be played out immediately. Furthermore, jitter may distort the relationship between media-units of a synchronization group. Hence, *intra-stream synchronization* as well as *inter-stream synchronization* is disturbed. To smoothen out the effects of jitter, media-units must be delayed at the sink such that a continuous playback can be guaranteed. For this purpose, *playout buffers* are required.

The main point addressed by model 2 is intra- and inter-stream synchronization and the calculation of the required buffer space. First, we regard the synchronization of a single substream. Based on a rule of Santoso [14], we formulate a theorem that states a well defined playout time[4] for a substream such that intra-stream synchronization can be guaranteed. Smooth playout cannot be guaranteed if starting before the playout deadline. Starting at a later time would require more buffer space. Afterwards, we will extend our considerations to the synchronization of multiple substreams. The main idea in order to achieve inter-stream synchronization is to maintain intra-stream synchronization for each substream [8]. We begin with an extension of the model parameters used so far (c.f. Table 2).

Throughout this paper, we assume *bounded* jitter and we use the definition of jitter given by Rangan et al. [12] who define jitter as the difference between the maximum delay and the minimum delay.

$$\Delta_k = d_k^{max} - d_k^{min}, \quad \forall k \qquad (3)$$

$$\Delta^{max} = max\{\Delta_k | k \in \{0 \ldots n-1\}\} \qquad (4)$$

2.5.2 Synchronized Playout for a Single Substream

To guarantee the timely presentation of a single stream subject to jitter, it is necessary to buffer arriving media-units at the client to compensate the jitter. The buffer is emptied at a constant rate.

Santoso et al. [14] have already shown that the temporal relationship within one continuous media stream can be preserved by delaying the output of the first media-unit for $d_k^{max} - d_k^{min}$ seconds. Based on this theorem, the playout deadline is derived. The deadline given by Santoso (case (a)) can be lowered in some situations (case (b)).

Symbol	Description	Unit
m	substream or server index ($m \in \{0, \ldots, n-1\}$)	[mu/sec]
r	requested display ate of each substream at the client	
d_k^{max}	maximum delay for substream *k*	[sec]
d_k^{min}	minimum delay for substream *k*	[sec]
$\bar{d}_k$	average delay for substream *k*	[sec]
Δ_k	jitter for substream *k*	[sec]
Δ^{max}	maximum jitter of all substreams	[sec]
Δ_k^+	maximum upper deviation from $\bar{d}_k$ due to jitter for substream *k*	[sec]
Δ_k^-	maximum lower deviation from $\bar{d}_k$ due to jitter for substream *k*	[sec]
Δ^{max+}	maximum upper deviation of all substreams	[sec]

Table 2. Model parameters

Theorem 1: Consider a single substream *k* in case of bounded jitter Δ_k given by (3). Then smooth playout can be guaranteed whenever either one of the following starting conditions holds true.
(a) $d_k^{max} - d_k^{min} = \Delta_k$ seconds elapsed after the arrival of the first media-unit, or
(b) the $(\lceil \Delta_k \cdot r \rceil + 1)$*-th* media-unit has arrived.
Proof: See [7]

When using the shifting strategy, we need to provide for sub-stream *k* a total buffer b_k of (for the derivation see [7]).

$$b_k = \left\lceil \left(2 \cdot \Delta_k + \Delta^{max+} - \Delta_k^+ \right) \cdot r \right\rceil \qquad (5)$$

3. Model 3: Resynchronization

3.1 Introduction

Models 1 and 2 assured both intra-stream synchronization and inter-stream synchronization under the assumption that jitter is bounded. In ATM based networks, this assumptions typically holds true at least for the network because we can express the acceptable QoS in parameters

[4] The playout time or playout deadline is defined as the time elapsed at the client between arrival and playout of the first media-unit of a substream.

like throughput, delay, jitter or cell losses [3]. If the end-system is not using a real-time operating system, bounded jitter can not be guaranteed.

When jitter is unbounded, an application needs to make certain assumptions on the amount of jitter since buffer space may be limited or the increase in end-to-end delay by too large a buffer is unacceptable [3]. To avoid buffer overflow in case of unbounded jitter, we introduce model 3.

Model 3 can be characterized as a scheme for resynchronization. We apply the concept of a **buffer level control** to detect asynchrony. To recover from asynchrony, we use feedback messages to the servers. Model 3 copes with asynchronies introduced by:

- Alteration of the average delay
- Clock drift
- Server drop outs

An *alteration of the average delay* leads to a **gap**[5] or a **concentration** in the continuous media stream. A gap occurs when the average delay becomes longer, a concentration can be observed when the average delay becomes smaller. The result of *clock drift* is very similar to the result of a change in delay, but arises much more slowly. Clock drift introduces a skew.

A mechanism is needed to adapt to changing conditions in order to preserve synchronization without allocating additional buffer space. Solving the problem by additional buffering based on worst case estimates might turn out to be a difficult task because changing conditions are unpredictable. Even if we succeed to get worst case estimates, we have to be aware that, first, resources are limited and that, second, large playout buffers increase the overall end-to-end delay which is not desired. Furthermore, uncontrolled buffering compensates the problems to a certain amount but will not resolve them over a long period of time.

Since all the described disturbing factors affect the buffer level, the buffer level can be regarded as an indicator for upcoming synchronization problems. Once a sink has discovered a problem, it has to take measures to restore synchronization. Since asynchrony is basically a shifting in the media stream, we only need to correct this shifting. Corrective actions must be feed back either to the source or to the sink in order to restore synchrony. The idea of taking the buffer level as an indicator is often referred to as **buffer level control**. Basic work in this area can be found in [13], [9] and [10]. Our model will uses some of their basic ideas and extends them to an applicable solution for the synchronization problem. In contrast to the previous work, we take model 1 and 2 as a basis for synchronization and extend them with a buffer level control. We focus mainly on buffer requirements and parameter tuning.

[5]The effect of a server drop out is also a gap in the media stream.

The next section examines models 1 and 2 with respect to a buffer level control and presents a buffer model suitable to realize a buffer level control. Finally, we discuss the tuning of the model parameters.

3.2 Buffer Level Control

3.2.1 System Model

The concept of buffer level control is often referred to as a *control loop* [9]. Sources transfer media-units over the network that arrive at the sink where they are buffered before playout. The current buffer level is periodically measured, and if an ill buffer level is found, the appropriate steps are taken. Actions may affect either the buffer itself or the server. In the former case, the loop is placed in the client, in the latter case it includes the client, the server and the network. Koehler et. al and Rothermel et al. [9], [13] propose a synchronization scheme that does not adapt the playout behavior of the server. Actions are taken exclusively at the sink whether by changing the consumption rate or by skipping/pausing. This kind of control loop compensates for disturbances to a certain amount depending on the allocated, available buffer space but sacrifices the real-time stream continuity.

We adopt to a concept where all components of the video server architecture are included in the control loop similar to the approach of Cen et al. [2]. As shown in figure 2, the architecture applies feedback actions to the sources via control messages in order to maintain synchronization at the sink.

(a) Feedback Filter

The **buffer level** for substream k at time t is denoted by q_{tk}. This value is periodically passed to a **filtering function** $S(q_{tk})$ to filter short-term fluctuations caused by jitter and to compute the **smoothed buffer level** $\bar{b}_{tk}$. Examples for filtering functions are the geometric weighting smoothing function (with α as smoothing factor) [13], [2], [11]:

$$\bar{b}_{tk} = S(q_{tk}) = \alpha \cdot \bar{b}_{t-1k} + (1-\alpha) \cdot q_{tk} \quad (\text{ with } \alpha \in [0,1]) ,$$

The main goal of filtering is to distinguish between buffer level changes caused by jitter and long-term disturbances. If the filter is too sensitive, or no filter is used at all, jitter causes actions for resynchronization although no exceptional situation has occurred. On the other hand, a filter that reacts to slowly to changing conditions takes actions too late with the result of a longer period of buffer starvation or overflow. Thus, presentation quality suffers.

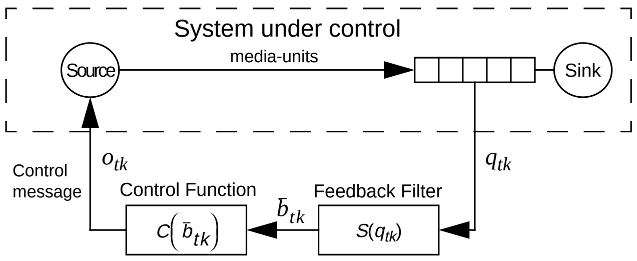

Figure 2. System model for the buffer level control [2]

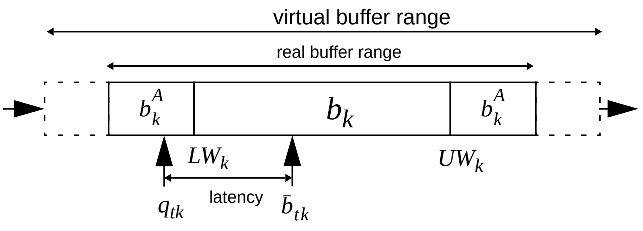

Figure 3. Buffer model with virtual and real buffers

(b) Control Function

The smoothed buffer level $\bar{b}_{tk}$ is passed to a **control function** $C(\bar{b}_{tk})$ that takes appropriate actions. For each substream buffer, a **lower water mark** LW_k and an **upper water mark** UW_k are defined. When $\bar{b}_{tk}$ falls below LW_k or exceeds UW_k, there arises the risk of starvation or overflow, respectively, producing an asynchrony. If this happens, a resynchronization or adaptation phase is entered whose purpose is to move $\bar{b}_{tk}$ back into between LW_k and UW_k. Depending on the extent of asynchrony, the control function sends an **offset** o_{tk} to the source. The source either skips the number of media-units specified in the offset or pauses for a duration of o_{tk} media-units. We prefer this technique over an alteration of scheduling speed, respectively production rate, at the source because we think the latter is too resource demanding and the QoS of other clients serviced by the server might suffer.

The sink stays in its resynchronization phase for a time R in order to let the smoothed buffer level react on the taken measures. At the end of the resynchronization phase $C(\bar{b}_{tk})$ controls again whether or not the buffer level $\bar{b}_{tk}$ has moved back in the normal area into between LW_k and UW_k. If not, a new resynchronization phase is [13] started.

3.2.2 Buffer Requirements

Models 1 and 2 provide the buffer space b_k needed to compensate jitter ([7]). In the following, we will denote b_k as a **kernel buffer**. Applying a buffer level control only to this buffer is not sufficient since each buffer level within the range of b_k must be regarded as normal due to the jitter effects. We fix LW_k and UW_k to 1 and b_k, respectively. To realize a buffer level control, we must admit buffer levels below and above the watermarks. Otherwise, it is impossible to get the smoothed buffer level $\bar{b}_{tk}$ below or above the watermarks.

We suggest the scheme of a so-called **virtual buffer** as indicated in figure 3 by the dashed lines. The virtual buffer includes at least the real buffer comprising the kernel buffer b_k and an **additional buffer** b_k^A. The virtual buffer is exclusively used for the calculation of buffer levels below and above the real buffer. This allows for a faster reaction of the smoothing function $S(q_{tk})$. The mapping between the real buffer level and the **virtual buffer level** q_{tk} is performed as follows:

- If neither buffer starvation nor buffer overflow occurs, the real buffer level equals the virtual buffer level.
- If a buffer overflow occurs, then the virtual buffer is increased for each discarded media-unit while the real buffer level remains unchanged.
- If a buffer starvation occurs, then the virtual buffer is decreased each time when the client finds an empty buffer while the real buffer level remains unchanged.
- If the normal state of the real buffer is restored by resynchronization measures, the virtual buffer level is reset to the real buffer level.

The size of b_k^A strongly influences the gracefulness of the resynchronization[6]. The smoothened buffer level $\bar{b}_{tk}$ always has a latency (see figure 3) compared with the virtual buffer level q_{tk}, i.e. q_{tk} might be below LW_k while $\bar{b}_{tk}$ still needs some time to fall below. Let $b_k^A = 0$, for instance. Then a buffer starvation occurs before it is recognized by the control function. Hence, presentation quality suffers depending on the value of b_k^A. We consider the following three cases for the size of b_k^A.

- Selecting $b_k^A = 0$ yields no gracefulness at all. Asynchrony immediately affects presentation quality and is soon discovered by a viewer.
- b_k^A can be dimensioned such that at least the period between the rise of asynchrony and the discovery by the control function is covered.
- For full gracefulness, b_k^A has to be chosen such that asynchrony does not affect presentation at all. The buffer space has to cover the period between rise, discovery and removal of asynchrony.

[6]Notice that the start-up latency is also influenced by the size of b_k^A. The larger b_k^A is, the longer it takes until the first media unit of a substream is played out because of the buffer level must exceed LW_k before the playout deadline given by model 2 can be applied.

3.2.3 Parameter Tuning

In our model, we have several parameters that must be chosen appropriately in order to trade-off reactiveness and overhead.

(a) Smoothing Parameter α

Obviously the latency of reaction to an asynchrony problem depends strongly on the behavior of $S(q_{tk})$. The more indolently $S(q_{tk})$ reacts, the later a resynchronization phase is entered, the more buffer space b_k^A may be desired to compensate for asynchrony as much as possible. On the other hand, the more sensitively $S(q_{tk})$ reacts, the more often resynchronization is done unnecessarily (due to the effect of jitter), the less buffer space b_k^A is needed to provide sufficient gracefulness. Hence, the tuning of $S(q_{tk})$ needs to trade-off between stability and reactivity. The choice of $S(q_{tk})$, respectively, helps to determine the additional buffer space b_k^A.

For further consideration we examine the filtering function given by () with respect to second case described above, i.e. the size of b_k^A must cover the period between rise and discovery of an asynchrony. This case is most interesting because it is influenced by $S(q_{tk})$. The behavior of the filter is determined by the parameter α:

- A large value of α yields strong smoothing, a stronger consideration of the past, and a more indolent reaction.
- A small value of α yields weak smoothing, a stronger consideration of the present, and a more sensitive reaction.

An upper bound for the choice of α is given by the available memory. A lower bound should be chosen such that starvation/overflow events due to jitter can be distinguished from long term disturbances. Accordingly, α should be set as high as possible while considering the buffer available.

In our experiments (see [6] for details) we found that a value of 0.6 or 0.7 for α is a good compromise with respect to the buffer requirement and the number of necessary resynchronization actions.

(b) Degree o_{tk} of Resynchronization

Resynchronization is performed by sending an offset to the servers to move the buffer pointer $\bar{b}_{tk}$ back into the area between UW_k and LW_k. The size of the offset o_{tk} can be determined by two different strategies: **fixed offset** or **variable offset**.

Employing the fixed offset strategy, o_{tk} is set to a constant value. Resynchronization is done slowly in subsequent resynchronization phases until synchronization is restored. The value should not be chosen too high because resynchronization, e.g. due to clock drift, is in the range of one or several media-units. High values could lead to oscillation.

When applying the variable offset strategy, o_{tk} varies depending on the extent of the occurred asynchrony. Notice that when applying the variable offset strategy several resynchronization phases could be needed as well because at the time when the offset is calculated (determined by the filtering function) the total extent of asynchrony might not yet be recognized. Nonetheless, synchronization is generally restored faster with a variable offset. We will present some experimental results that compare both strategies.

(c) Duration R of Resynchronization

The duration of a resynchronization phase is defined by R. After R seconds the control function once more compares the smoothed buffer level with the watermarks. Again, resynchronization actions may be taken.

R must be chosen sufficiently large that the server can perform the resynchronization, that is, the action must already have taken effect on the client. Selecting R too small leads to numerous unnecessary resynchronization phases where during each phase the extent of asynchrony is overestimated. Low values of R can result in oscillation. For large values of R several resynchronization phases are needed as well but the total time of resynchronization can become unacceptably long. So, in both cases presentation quality might be strongly influenced.

3.3 Experimental Results

Based on the prototype implementation of the Video Server Array we have implemented the proposed synchronization scheme for evaluation purposes. For implementation details, refer to [1] and [6].

The following experiments have been performed on a dedicated SUN Sparc 10 workstation as a client. We used two videos, each one distributed across two servers:

- A “Bitburger” commercial, sampled at a rate of 16 fps (frames/sec) with a total length of 462 frames[7].
- A scene from the production “Seaquest”, sampled at a rate of 16 fps with a total length of 6710 frames.

We evaluated the efficiency of the buffer level control mechanism. The prototype of the Video Server Array is implemented in an ATM-LAN environment. So we faced the problem that events like gaps or concentrations within a stream are rather unlikely. Thus, we simulated these events in the servers. The amount of asynchrony can be

[7] in the context of video streams we use the term frame to denote a media-unit.

specified by the user upon starting a server. The server then periodically introduces drop outs in scheduling or sends several frames at once. The client attempts to resynchronize the server by sending back offsets. The following parameters have been used:

- Smoothing factor for the geometric weighting function: $\alpha = 0.7$
- Amount of injected asynchrony[8]: -8, -4, +4, +8 [frames]
- Resynchronization strategy: *fixed offset* and *variable offset*

The variable offset was calculated by taking the difference between q_{tk} and the watermarks. The fixed offset was set constant to 1. We allocated two buffer slots for the substream. This corresponds to the kernel buffer b_k. Furthermore, for the additional buffering b_k^A, we were using three buffer slots each, above and below b_k. Consider figure 4, showing the virtual buffer level and the filtered buffer level over time for the resynchronization of a concentration of eight frames. The y-axis shows the virtual buffer level while the x-axis denotes the consumption period. The upper bound (watermark) of the real buffer level is denoted by b while the lower bound is not shown in the figure. Thus, b_k^A equals $b - UW$ and b_k is given by $UW - LW$. The virtual buffer level ranges from 1 to 108 because we arbitrarily selected a number of 50 frames above and below the real buffer to calculate the virtual buffer. Figure 4 shows the course of resynchronization for the fixed offset strategy.

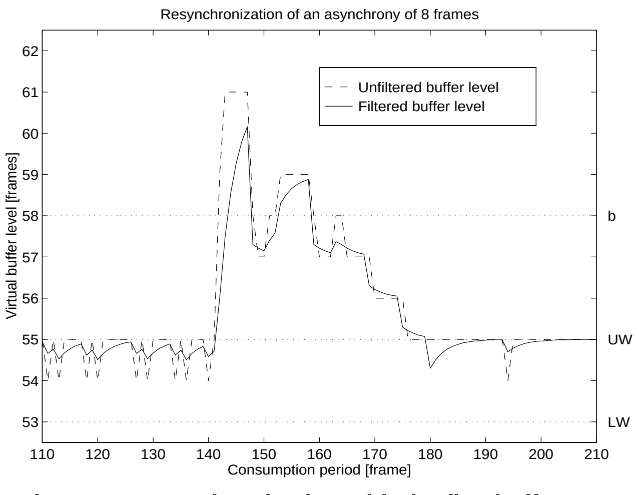

Figure 4. Resynchronization with the fixed offset

The first resynchronization phase is entered exactly during consumption period 142 when the filtered buffer level crosses the upper watermark *UW*. The virtual buffer level rises up to 61, that is, four frames are discarded. The client then sends an offset of -1 to the server. The client undergoes 7 subsequent resynchronization phases at the whole. These phases are indicated by the peaks. Synchronization is restored exactly during consumption period 180 when the filtered buffer level falls below *UW*.

We now consider the same situation with the variable offset strategy. The course of the filtered and unfiltered buffer level is depicted in figure 5.

Resynchronization starts during consumption period 130. Again, a number of four frames is discarded. The client first sends an offset of -3 frames to the server. Already after this resynchronization action, the buffer level falls below UW for a short period of time. Now, two additional resynchronization phases are needed until synchrony is restored. In each phase an offset of -2 is sent to the server. Synchronization is exactly restored during consumption period 149. In contrast to the fixed offset strategy, only 19 frame periods are needed to regain synchrony. The results also show clearly that resynchronization with a variable offset becomes even more efficient for larger asynchronies because the adoption is performed faster.

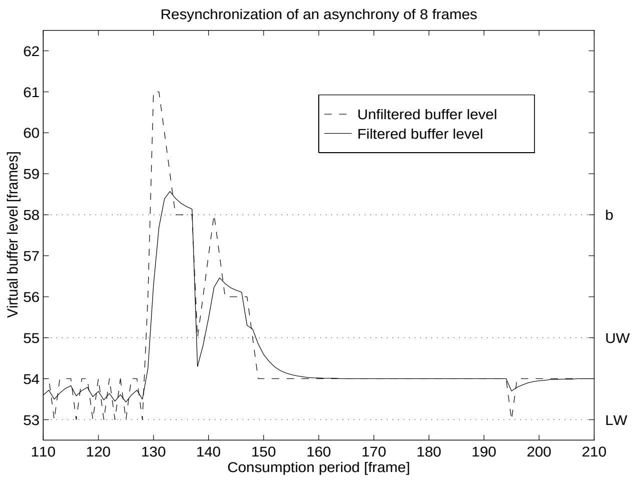

Figure 5. Resynchronization with variable offset

4. Conclusion

We have presented a scheme for intra- and inter-stream synchronization of distributed stored multimedia streams. Our scheme comprises three models that assure synchronization in an environment with different delays, jitter, server drop-outs, clock drift, and an alteration of the average delay. The mechanisms described do not rely on synchronized clocks within the network. In contrast to existing synchronization solutions, the scheme is suitable for streams that are striped across multiple server nodes as well as for a single server approach.

[8]Negative values denote a drop out while positive values denote a concentration.

The scheme presented has been successfully implemented in our video server prototype [1] where each video is distributed (striped) over *n* server nodes.

Acknowledgment. The work described in this paper was supported by the Siemens Nixdorf AG, Munich.

5. References

[1] C. Bernhardt and E. W. Biersack. The server array: A scalable video server architecture. In W. Effelsberg, A. Danthine, D. Ferrari, and O. Spaniol, editors, *High-Speed Networks for Multimedia Applications*. Kluwer Publishers, Amsterdam, The Netherlands, 1996.

[2] S. Cen, C. Pu, R. Staehli, C. Cowan, and J. Walpole. A distributed real-time MPEG video audio player. In T. D. C. Little and R. Gusella, editors, *Proceedings of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV'95)*, volume 1018 of *LNCS*, pages 142–153, Durham, NH, April 1995. Springer Verlag, Heidelberg, Germany.

[3] J. S. Cormac. *Synchronisation Services for Digital Continuous Media*. PhD thesis, University of Cambridge, Cambridge, England, October 1992.

[4] W. Effelsberg, T. Meyer, and R. Steinmetz. A Taxonomy on Multimedia-Synchronization. In *Proceedings of the Fourth Workshop on Future Trends of Distributed Computing Systems, Lisbon, Portugal, Sep. 1993*, pages 97–103. Eyrolles, 1993.

[5] J. Escobar, C. Patridge, and D. Deutsch. Flow synchronization protocol. *ACM Transactions on Networking*, 2(2):111–121, April 1994.

[6] W. Geyer. Stream synchronisation in a scalable video server array. Master's thesis, Institut Eurecom, Sophia Antipolis, France, September 1995.

[7] W. Geyer, C. Bernhardt, and E. Biersack. A synchronization scheme for stored multimedia streams. In B. Butscher, E. Moeller, and H. Pusch, editors, *Interactive Distributed Multimedia Systems and Services (European Workshop IDMS'96, Berlin, Germany)*, volume 1045 of *LNCS*, pages 277–295. Springer Verlag, Heidelberg, Germany, Mar. 1996.

[8] Y. Ishibashi and S. Tasaka. A synchronization mechanism for continuous media in multimedia communications. In *IEEE Infocom'95*, volume 3, pages 1010–1019, Boston, Massachusetts, April 1995.

[9] D. Koehler and H. Mueller. Multimedia playout synchronization using buffer level control. In *2nd International Workshop on Advanced Teleservices and High-Speed Communication Architectures*, pages 165–180, Heidelberg, Germany, September 1994.

[10] T. D. C. Little and F. Kao. An intermediate skew control system for multimedia data presentation. In *Proceedings of the 3rd International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 121–132, San Diego, CA, November 1992.

[11] H. Massalin and C. Pu. Fine-grain adaptive scheduling using feedback. *Computing System*, 3(1):139–173, 1990.

[12] P. V. Rangan, H. M. Vin, and S. Ramanathan. Designing an on-demand multimedia service. *IEEE Communications Magazine*, 30(7):56–65, July 1992.

[13] K. Rothermel and T. Helbig. An adaptive stream synchronization protocol. In T. D. C. Little and R. Gusella, editors, *5th International Workshop on Network and Operating System Support for Digital Audio and Video*, volume 1018 of *LNCS*, Durham, New Hampshire, USA, April 1995. Springer Verlag, Heidelberg, Germany.

[14] H. Santoso, L. Dairaine, S. Fdida, and E. Horlait. Preserving temporal signature: A way to convey time constrained flows. In *IEEE Globecom*, pages 872 – 876, December 1993.

[15] R. Steinmetz and K. Nahrstedt. *Multimedia: Computing, Communications and Applications*. Innovative Technology Series. Prentice Hall, Englewood Cliffs, NJ, 1995.