Stream Synchronization in a Scalable Video Server Array

Werner Geyer

Technical report September 15, 1995

Institut Eurécom
2229, route des Crêtes
Sophia Antipolis
F-06904 Sophia Antipolis Cedex, France
geyer@eurecom.fr

Directed by
Prof. Dr. Ernst Biersack and
Dipl.-Inform. Christoph Bernhardt

Abstract

The design of video servers differs significantly from that of traditional storage and retrieval servers. Video servers must satisfy the tight temporal constraints that stem from the continuous nature of audio and video. Large transfer rates and storage capacity are required. Since video servers are thought to offer multimedia services to a large gamut of applications they have to be highly scalable. A novel video server architecture called *video server array* provides better scalability properties and optimal load-balancing compared to traditional video servers. A single multimedia *stream* is distributed across multiple server nodes. During retrieval of the stream all involved server nodes are equally utilized. Server nodes deliver independent *substreams* of media units that are recombined at the client. In the context of this architecture, the thesis presents a scheme to ensure the continuous and synchronous delivery of data to the client. We propose a protocol to initiate the playback of a stream in a synchronized manner and we derive buffer requirements in order to maintain both the continuity within a single substream and the synchronization between related substreams. Furthermore, we employ the concept of a buffer level control in order to detect asynchrony during retrieval, and to regain synchronization. Experimental results prove the effectiveness of the proposed scheme that has been implemented in a prototype of the video server array.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

codec	coding/decoding
fps	frames per second
ms	milliseconds
2PC	Two Phase Commit
AAL	ATM Adaptation Layer
ATM	Asynchronous Transfer Mode
AV	Audio/Video
B-ISDN	Broadband Integrated Services Digital Network
EDF	Earliest-Deadline-First
FCFS	First-Come-First-Served
FDDI	Fiber Distributed Data Interface
FIFO	First-In-First-Out
GCD	Greatest Common Divisor
GUI	Graphical User Interface
I/O	Input/Output
IP	Internet Protocol
JPEG	Joint Photographic Expert Group
LAN	Local Area Network
LTS	Logical Time System
LW	Lower Watermark
MPEG	Motion Pictures Expert Group
NTP	Network Time Protocol
PDU	Protocol Data Unit
QoS	Quality of Service
RAID	Redundant Array of Inexpensive Disks
RTS	Relative Time Stamp
Tcl/Tk	Tool command language and the Toolkit
TCP	Transmission Control Protocol

UDP	User Datagram Protocol
UW	Upper Watermark
VCR	Video Cassette Recorder
VHS	Video Home System

1 Introduction

Advances in communication technology give rise to new applications in the domain of multimedia. Emerging high-speed, fiber-optic networks have made it feasible for multimedia applications such as Video On-Demand, Tele-Shopping or Distance Learning to get on-line access to a variety of information. These applications typically integrate different kind of media such as audio, video, text or images. Customers of such a service will be permitted to retrieve the digitally stored media from a server for real-time playback. The storage facility providing these multimedia services is commonly called *video server* [Ber95b].

The design of a video server differs significantly from traditional storage and retrieval services. First, the tight temporal constraints imposed by the continuous nature of audio and video have to be satisfied. Audio and video streams consist of sequences of media quanta (continuous media stream) which convey information only if presented continuously in time. This is in contrast to a textual object, for which spatial continuity is sufficient. Furthermore, the playback of multiple media streams constituting a multimedia object should not only be continuous, but also temporally coordinated. Traditional file servers cannot give guarantees on the timely delivery of data. [Ran92], [Gem95]

Second, the handling of multimedia data is very demanding with regard to transmission bandwidth and storage capacity. Digital video and audio is played back at a very high rate. Thus, scarce resources should be used efficiently and the overall workload has to be distributed equally over all components of a server. A video server must provide efficient mechanisms for storing, retrieving and manipulating large amounts of data at a high speed. [Gem95]

Third, a video server should be designed scalable so as to allow a smooth adjustment to a growing demand for multimedia services. Scalability allows for the expansion of the video server's capacity, for instance by introducing additional servers in an existing architecture without changing software or underlying networks. Furthermore, different size video servers are required because there exists a wide range of multimedia applica-

tions. [Ber95a]

A new video server architecture where the entire video server consists of several servers grouped in a *server array* has been proposed by Bernhardt [Ber95a]. The architecture has good scalability properties and allows for an optimal load balancing. The so-called *Video Server Array* works similar to a RAID array. A single multimedia stream is distributed across multiple server nodes. Each node stores only a subset of the entire stream. During retrieval each server contributes with its portion to the entire stream. The subsets have to be sent in a coordinated manner to the client who has requested the service. Each subset constitutes a so-called *substream.* [Ber95a]

The challenge associated with this architecture is to coordinate the presentation of a media stream such that both the continuum of each substream and the temporal relationship between substreams are maintained in order to offer a transparent service to the user. The timely coordination of multimedia streams is called *synchronization.*

This diploma thesis presents a scheme that solves the problem of synchronization with respect to the architecture of the Video Server Array. We propose a synchronization approach based on three models. Each model covers certain issues of the synchronization problem in the given architecture. Furthermore, we developed a prototype implementation of the Video Server Array so as to prove the feasibility and performance of this architecture. Within this prototype we implemented the proposed synchronization scheme, and we evaluated its performance in a real system environment.

The remainder of the thesis is structured as follows. The second Chapter gives a detailed description of the architecture of the scalable Video Server Array. Differences and benefits in contrast to traditional approaches are shown.

Throughout the third Chapter, general synchronization topics in multimedia are introduced. We give a characterization of the term synchronization, and requirements with respect to QoS of synchronization are discussed. Furthermore, we give a list of criteria in order to classify the area of synchronization followed by a survey and classification of existing synchronization solutions.

The synchronization approach is presented in Chapter four. We introduce the basic concepts and assumptions of the scheme. The causes of asynchrony are discussed, and the synchronization problem at hand is classified. We describe the three synchronization models. Model 1 covers the problem of initiating the playback of a media stream in a synchronized fashion. Based on the results of the first model we next consider the prob-

lem of assuring a smooth playout during the playback of a stream. As soon as the synchronization of a stream is lost it has to be resynchronized. Model 3 presents a mechanism to handle the problem of resynchronization. Finally, we present some experimental results gained in the prototype implementation.

Chapter five describes the prototype implementation of the Video Server Array. We give an overview of the system architecture, followed by a description of the protocol flow between the involved software components. Afterwards, the design of each component of the prototype, namely *server*, *client,* and *metaserver*, is presented.

The work is concluded with a brief review and a discussion of the efficiency of the proposed synchronization scheme.

2 The Scalable Video Server Array

2.1 Autonomous Video Server

A typical scenario for a multimedia on-demand service consists of several video servers that are connected to clients or customers, respectively, via a high-speed network like ATM or FDDI. The video server is the network element providing the source of multimedia material, which can be requested by a customer [Del94]. The requested material is stored on extremely high-capacity secondary storage media like hard disks and on tertiary storage devices as well [Fed94].

The most critical element in a multimedia on-demand architecture is the video server that supports continuous retrieval of media information from disks [Gem95]. While emerging high-speed networks may provide high bandwidth at modest costs, the I/O system of a video server becomes a bottleneck. We must always keep in mind that such systems are meant to service several up to thousands of customers [Ber95a]. Thus, a video server must be scalable with respect to the number of streams that can be serviced concurrently and with respect to the amount of stored material. The scalability of a single powerful video server is ultimately limited by its system bus bandwidth. The amount of storage and the disk bandwidth can be easily extended just by introducing new disks that are grouped in a RAID. Within a RAID, data is scattered equally across the set of disks and retrieved in parallel. Nonetheless, if the bandwidth demand exceeds the bandwidth of the system bus, adding new disk bandwidth yields no improvement in the possible number of streams that can be serviced simultaneously. To overcome this limitation, additional server nodes have to be added to the multimedia on-demand architecture. Applying multiple servers to the task of video delivery is often referred to as *server-level parallelism* [Fed94].

Traditionally such a configuration consists of several *autonomous video servers* placed on the network so as to improve the peak aggregate bandwidth. Since video material is stored in its entirety onto the server nodes, it must be decided which streams are stored on which server. If many people simultaneously request a video that is stored on only

one of the servers, the system suffers from load balancing problems, this is, one server is overloaded while the others sit idle [Fed94]. Thus, the distribution of video material across the nodes has to be constructed carefully so as to avoid so-called *hot-spot servers* as indicated in figure 1. On the other hand, merely replicating popular videos across several servers results in poor storage utilization. The problem is complicated due to the fact that the popularity of a stream is not known *a priori* or might change over time. Consequently, complex monitoring systems are needed to reorganize the distribution whenever the popularity of the stored material changes [Ber95a]. We can conclude that the traditional design of video servers suffers from load balancing problems and/or poor storage utilization.

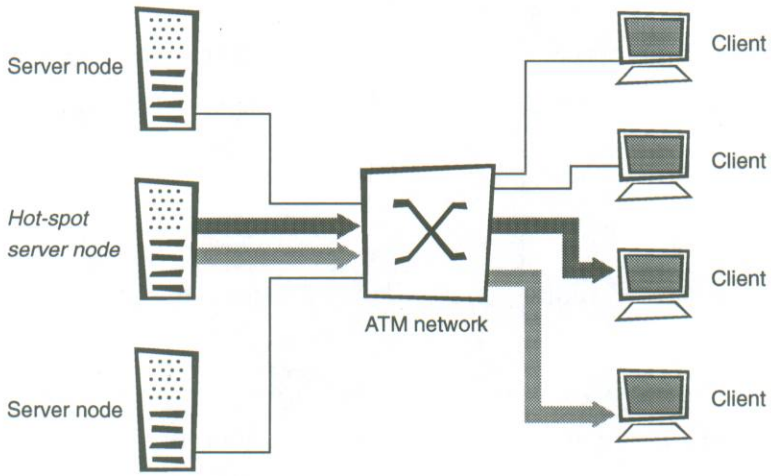

Figure 1: Set of Single, Autonomous Video Servers.

2.2 Video Server Array

A novel architecture for a scalable video server called *Video Server Array* proposed by Bernhardt et al. [Ber94] is also based on server-level parallelism but overcomes the problems of load balancing, scalability, and storage utilization. Server nodes are configured into an array similar to a RAID. In contrast to the autonomous concept, video material is not stored in its entirety on the nodes. A single video stream consisting of a sequence of media units is distributed (*striped*) over several server nodes of the server array. During retrieval of the video stream all server nodes involved are equally utilized. Each one of the servers delivers an independent continuous media stream called *substream*. At client site substreams are recombined. The load balancing is optimal if each stream is distributed over all existing server nodes of the array as indicated in figure 2 [Ber95a]. The load

during the retrieval of two substreams is equally distributed over the servers and the network connections to the servers. In the traditional video server architecture the two videos might happen to be stored on the same server as demonstrated in figure 1. The server and the network connections are hot spots in the overall server configuration. [Ber95b]

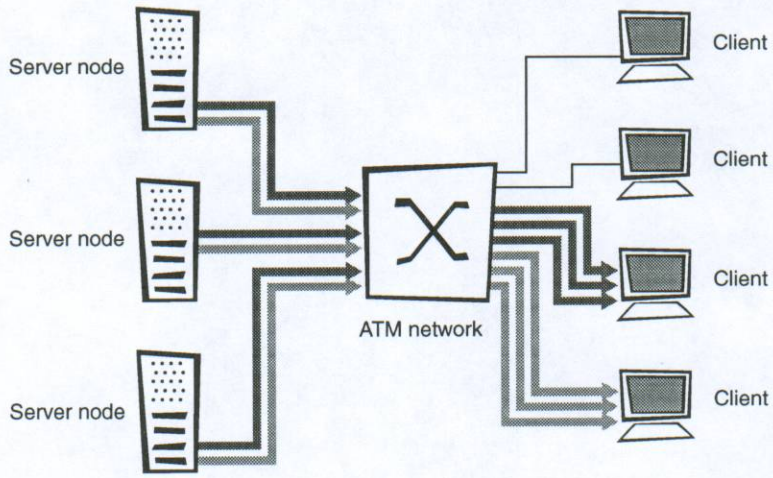

Figure 2: Striping in the Video Server Array.

In contrast to traditional video servers, scalability properties of the server array are better because just another server node must be added. After adding a new node, the distribution of the substreams must be reorganized once in order take advantage of the added bandwidth and capacity, i.e. the video material must be striped onto to server nodes again. The redistribution may be performed off-line or during periods of little demand for multimedia services [Den95].

Further optimization of the architecture can be achieved by the choice of the *striping block size* that defines the number of contiguous frames that is stored on a single server node [Ber95b]. If employing *inter-frame striping,* each frame is stored in its entirety onto one of the server nodes. This technique is also called *single frame striping*. Usually digital material is stored in compressed form, resulting in different sizes of frames. So, load balancing becomes sub optimal depending on the actual distribution of the data.

Load balancing can be improved by *intra-frame striping*, also known as *subframe striping*. Each frame is divided into *n* sub frames where n denotes the number of servers in the server array. This assures that the total amount of video data is distributed equally over the nodes of the server array. The choice of the striping block size determines the synchronization requirements in the video server array as we show in section 4.

3 Multimedia Synchronization

3.1 Overview

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 conveyed by these streams a certain temporal relationship exists. Multimedia systems must guarantee 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* [Eff93].

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 [Rot95a]. The classical example of stream synchronization is the *lip-synchronized* presentation of audio and video [Esc94].
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. As 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.
Thus, to solve the problem of stream synchronization we have to regard both issues which are tightly coupled [Blu94].

3.1.1 Intra-Stream Synchronization

A continuous media stream consists of a sequence of (encoded)[1] samples which are transferred between source and sink. The task of intra-stream synchronization often

referred to as *serial synchronization* [Bul91] is to maintain the inherent temporal properties given by the sampling rate. Hence, serial synchronization has to reproduce information as originally captured. There exist exceptions, of course, when, for instance, the playback rate is altered to achieve VCR functions like fast-forward. [Cor92]
The temporal relationship within a single stream is mainly disturbed for the following reasons [Blu94], [Lit92], [Cor92]:

- *Network jitter*
- *End-system jitter*
- *Clock drift*
- *Changing network conditions*

Network jitter denotes the varying delay that stream packets experience on their way from the sender to the receiver network I/O device. It is introduced by the buffering in intermediate nodes. *End-system jitter* refers to the variable delays arising within the end-systems, and is caused by varying system load and the packetising and depacketising of frames with variable size, that are passed through the different protocol layers. Jitter is commonly equalized by the use of an *elastic buffer* at the sink site [Blu94].

Capturing, reproduction and presentation of continuous media is driven by end-system clocks. In general, clocks cannot be assumed to be synchronized. Due to temperature differences or imperfections in the crystal clock the frequency of end-system clocks can differ over a long period of time. The result is an offset in frequency to real time and to other clocks which causes a drift rate from 10^{-6} sec/sec up to 10^{-3} sec/sec.[2] The problem of clock drift can be coped with by using time synchronizing protocols within a network. The *network time protocol* (NTP), for instance, offers a global (virtual) time to its service users. Otherwise, if the problem of clock drift is neglected, buffer overflow or buffer starvation on the client site will arise over a long period of time [Cor92], [Cor95]. The effect of clock drift is also known as skew that is defined as an average jitter over a time interval [Lit92].

Changing network conditions, not introduced by jitter, refers to a variation of connection properties, for instance an alteration of the average delay or an increasing rate of lost frames.[3] These effects strongly depend on the QoS the underlying network can provide.

[1] Commonly used digital compression techniques are, for instance, JPEG, MPEG, or H.261.

[2] A drift of 10^{-6} sec/sec is a more common value for today's systems. If the problem of synchronization is restricted to the playback of a single video, for instance with a duration of 90 min, a total asynchrony of 5.4 milliseconds will arise. This deviation can not be perceived by a user. On the other hand, a drift of 10^{-3} sec/sec resulting in an asynchrony of 5.4 seconds will strongly influence presentation quality. [Cor92]

However, synchronization mechanisms have to cope with this kind of problem.

3.1.2 Inter-Stream Synchronization

Inter-stream synchronization, also called *parallel synchronization* [Bul91], has to establish and maintain a certain temporal relationship between two or more related continuous media streams. Little et al. [Lit91] present thirteen possible relationships between time intervals of related streams. Common relations are interval *a equals b* or *a before b*. Inter-stream synchronization has to ensure these relations to a certain accuracy determined by the end user. A set of streams to be synchronized is often called *synchronization group* [Esc94]. All streams within the group are synchronized to each other according to the predefined temporal relationship. Different priorities can be assigned to streams within a group corresponding to their importance. The stream with the highest priority is called *master*. It dictates the synchronization of the remaining streams of the group called *slaves*. Again the classical example of lip-synchronization of audio and video: as the human perception of audio is much more sensitive to asynchrony one should assign the master role to the audio stream [Rot95b], [Ran93].

3.2 Requirements for Synchronization

In the previous section we already broached that accuracy of synchronization is or has to be determined by user requirements. Thus, synchronization mechanisms have to take these requirements into account.

The accuracy by which synchronization has to be done is mainly determined by the human perception. Recent experiments conducted by Steinmetz et al. [Ste93b] show that a mismatching in lip-synchronization as low as 120 ms between audio and video can be already perceived by the application user. On the other hand, playing out text (e.g. subtitles) up to 200 ms before the audio, is still well tolerable. The major goal of synchronization is to minimize this deviation or mismatching between related streams such that it is kept within defined *tolerance* bounds. [Li94]

Another dimension of synchronization is given by the kind of source. Literature distinguishes between *life synchronization* and *synthetic synchronization* [Ste93a]. In the former case capturing of samples and playback have to be performed almost at the same

---

[3] Commonly unconfirmed datagram services are used for transmitting multimedia data. Retransmissions are not suitable as data is extremely time critical. A datagram service is unreliable and frames are lost from time to time. The proposed synchronization scheme handles lost frames such that the last frame is displayed again, this is, the last frame is doubled.

time while in the latter case, samples are recorded, stored and played back later. Hence, for life synchronization, e.g. in teleconferencing, the delay must be kept as low as possible. Consequently, the size of the elastic buffer which is dictated of course to a certain amount by the jitter has to be minimized. For recorded media, constraints are not so restrictive; higher delays are tolerable, and the fact that sources can be influenced proves to be very advantageous as shown later in the proposed synchronization model. 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 buffer level as low as possible. [Koe94]

3.3 Existing Synchronization Solutions

No common classification scheme for synchronization approaches exists. At present synchronization mechanisms are either application specific or try to cover synchronization on a higher level independent of the application at hand. We give a collection of criteria that characterize various aspects of the generic synchronization problem. In the following section we pick up three of the most important criteria and attempt to classify existing synchronization approaches. The classification scheme is adopted from [Koe94].

3.3.1 Classification Criteria

- **Time**
  Schemes can be classified whether they assume a synchronized, global time within a network or not. *Global clocks* allow to compensate for clock drift and to calculate exact values for delay or jitter. On the other hand they are not available on every system at present, and accuracy is not always as fine-grained as desired. A sophisticated, complex protocol is required. NTP [Mil91] is an example for such a protocol in the internet environment.
- **Direction**
  The direction determines whether a mechanism is applicable for *intra-stream synchronization* or *inter-stream synchronization.* The terms are explained in section 3.1.
- **Location**
  Synchronization functionality may be located either on the source site or on the sink site, depending on the assumptions taken about the efficiency of the sink [Koe94].
- **Methods**
  Restoring synchronization can be done either by speeding up or slowing down presentation or production of media units, or by *stuffing*. The latter can be performed either by duplicating/deleting media units or by pausing/skipping, respectively. The effect of the methods is the same. By applying, a stream that is out of synchronization may either catch up or slow down so to regain synchronization. Changing speed

might not always be possible as resources, e.g. bandwidth, are limited. [Koe94]

- **Participants**
  Depending on the number of participating sources or sinks, the following synchronization sets can be distinguished [Esc94]:
  - *one-to-one*
  - *one-to-many*
  - *many-to-one*
  - *many-to-many*

  A similar classification taking into account whether synchronization is done locally or distributed is given by [Bul91].
- **Flexibility**
  Adaptive mechanisms react on changing network conditions. *Static* mechanisms assume a constant end-to-end delay.

3.3.2 Classification of Synchronization Solutions

A survey of multimedia synchronization mechanisms can be found in [Ehl94] and [Koe94]. Ehley distinguishes between distributed and local methods. Within these dimensions she summarizes different criteria like *protocol based* solutions or the kind of distribution among nodes. Each technique is described briefly. We adopt to Koehler as he clearly distinguishes between different dimensions. Existing work is classified within a *3D-cube*. Though one has to be aware that the dimensions are not completely independent. The classification is shown in figure 3. For the dimensions we have chosen time, method and location. Although direction is an important issue it has not been selected as most of the synchronization schemes cover both intra-stream and inter-stream synchronization.

The work of Steinmetz [Ste90a] and Little et al. [Lit90] is not classified in the cube because they present no concrete synchronization scheme but examine synchronization issues from a more abstract level. Steinmetz discusses characteristics of multimedia systems and presents a set of constructs to express intermedia relationships. Little models the intermedia timing based on timed Petri nets.

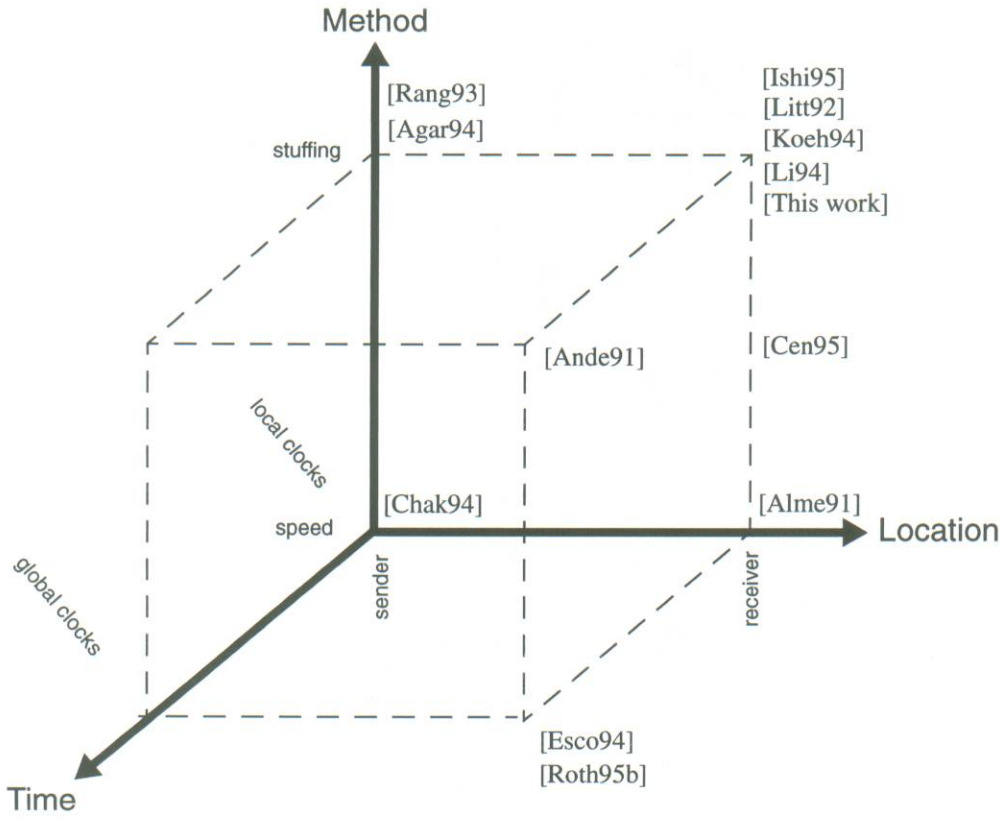

Figure 3: Classification of Synchronization Mechanisms [Koe94].

Escobar et al. [Esc94] and Rothermel et al. [Rot95b] assume a global synchronized time. The synchronization mechanism basically relies on determining the different kind of delays each stream experiences, using time stamps. At the receiver different delays are equalized to the maximum delay by buffering. Rothermel enhances this basic mechanism with a *buffer level control* and a *master-slave* concept. The usage of a logical time system (LTS) proposed by Anderson et al. [And91] is very similar to global clocks. He describes techniques to recover from asynchrony within a single-site workstation. Resynchronization is done by skipping and pausing, respectively.

Rangan et al. [Ran93] present a synchronization technique based on feedback mechanisms. Synchronization is done on sender site because less powerful receiver stations are assume that only have to send back the number of the currently displayed media unit. Asynchrony can be discovered by the use of so-called relative time stamps (RTS). Synchrony is restored by deleting or duplicating media units. Trigger packets are exchanged periodically so to calculate the relative time deviation between sender and receiver. Agarval et al. [Aga94] adopt to the idea of Rangan and enhance the scheme by dropping the assumption of bounded jitter.

Chakrabiti et al. [Cha94] also apply a feedback mechanism. The sender production rate is controlled by the used buffer space at the receiver site. A receiver clock determines a constant consumption rate from the buffer.

The technique of *phase locked loops* is usually applied to restore synchronization of continuous data transmitted via an asynchronous network, e.g. ATM. The buffer level on receiver site is compared to a nominal value. Basically, the read out clock is driven by the fill level. A phase locked loop mechanism is described by Almeida et al. [Alm91].

A more pragmatical solution to synchronization is given by Cen et al. [Cen95]. They describe the synchronization mechanism of a distributed MPEG player in the Internet environment. It is based on software feedbacks to the sender. Synchronization is regained either by influencing production speed or by skipping/pausing. To filter out jitter effects Cen employs low pass filters.

The synchronization concept of Li et al. [Li94] is based on a B-ISDN network which can give certain guarantees with respect to average delay and jitter. Based on these parameters a stream delivery schedule is calculated. Within the schedule certain tolerance parameters have to be fulfilled. If not, for instance due to statistical deviations, a fine grain tuning is done on the receiver site by adjusting the schedule. Little's *skew control system* [Lit92] applies a kind of buffer level control. Within a certain nominal buffer level inter-stream synchronization is maintained. When defined thresholds are reached, synchronization is regained by duplicating or dropping frames. Little assumes a constant playout rate and guaranteed network resources. The synchronization scheme of Koehler et al. [Koe94] covers intra-stream synchronization exclusively. The mechanism is based on a controlling the receiver buffer level. The fill level is filtered, compared to a nominal value and evaluated by a control function. Correspondingly, intra-stream synchronization is restored by duplicating or deleting media units in the local buffer.
Ishibashi et al. [Ish95] propose a time-stamp-based synchronization scheme suitable for both intra-stream synchronization and inter-stream synchronization. Taking the assumption of bounded jitter, synchronization is done similar to Escobar's approach of equalizing the delay. Intra-Stream synchronization is based on the theorem of Santoso [San93]. In order to perform synchronization in case of unknown delay Ishibashi applies a concept based on delay estimations. Once intra-stream synchronization is established, inter-stream synchronization can be maintained with a certain probability. Corrective actions are taken by skipping/pausing. The theory is based on the absence of clock drift.

We propose an application specific synchronization scheme for stored media suitable for

both intra- and inter-stream synchronization [This work]. 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. A classification of our approach can be seen in figure 3. Our work has been mainly influenced by the ideas of Ishibashi [Ish95] with resect to intra- and inter-stream synchronization. Based on a theorem of Santoso [San93] we derive buffer requirements and playout deadlines so to assure inter- and intra-stream synchronization. For the technique of resynchronization we adopt to a similar scheme as described by Koehler and Rothermel [Koe94], [Rot95c]. To initiate the playback of a stream in a synchronized manner we developed a *start-up protocol*. The synchronization scheme that consists of three models is described in the following chapter.

4 Synchronization in the Video Server Array

We present a synchronization mechanism that is based on three models. First, we introduce the assumptions on which the scheme is based. Then we identify sources of asynchrony in the *Video Server Array*, followed by a brief characterization of the synchronization problem at hand. Section 4.4 describes each model of the scheme. We close with some experimental results on the efficiency of the synchronization scheme.

4.1 Basic Concepts and Assumptions

The Video Server Array as described in section 2 is based on a distributed architecture consisting of several servers and clients connected via an ATM switch.[4] For the proposed synchronization scheme we replace the switch by an arbitrary ATM subnetwork. Figure 4 shows the underlying architecture for the synchronization problem.

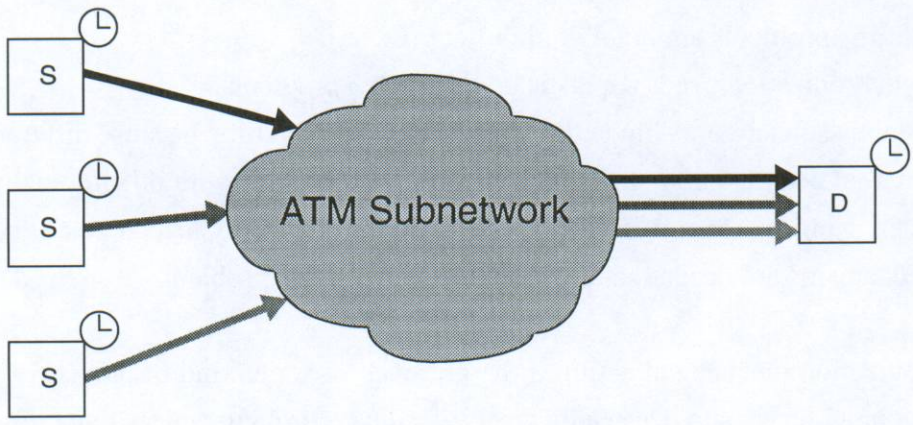

Figure 4: Distributed Architecture for the Synchronization Scheme.

Each of the servers denoted by *S* delivers an independent *substream* of media units. The

[4] For further consideration the terms server, source and sender are regarded equivalent as well as client, sink and destination.

production rate is driven by the server clock. Media units are transmitted to the Destination *D* via an ATM subnetwork. Transfer service is offered to the source and client application by a sort of datagram service (e.g. UDP/IP) layered on top of the ATM service. Arriving units are buffered in FIFO queues at the destination *D*. The playout of the entire *stream* composed out of the substreams is driven by the destination's clock.

The synchronization scheme is developed on the assumption of *inter-frame striping* (refer to section 2) with a striping block size identically to a frame. Frames are stored in their entirety onto the server nodes. Enhancements due to *intra-frame striping* are discussed briefly in section 4.4.6. Let n be the number of server nodes in the array and r be the frame rate of the entire stream. Frames are assumed to be distributed over the servers in a round-robin manner where the first server stores the first frame, the second server stores the second frame, etc. Each server contributes to the entire stream with a rate of r/n.

We further apply the concept of a *burst transfer* mode: frames are sent in periodic bursts according to the production rate. In contrast to the burst mode, in *continuous mode* the sending of data is distributed uniformly over the scheduling interval defined by the production rate [Ber95b].

The use of global, synchronized clocks within the network facilitates the problem of synchronization. However, the synchronization scheme does not assume the presence of a global time for several reasons: [Ran93]

- Network time protocols are not ubiquitous at present.
- Clock synchronization needs sophisticated, complex protocols.
- Heterogenous subnetworks hinder the presence of a global time because different organizational domains do not wish to synchronize their time with other domains.
- Even when using synchronized clocks, errors are introduced by inaccuracies and thus, mechanisms are needed in any case to cope with this problem.

The synchronization mechanism assumes the presence of some kind of *admission control* strategy on the server site. *Deterministic admission contro*l guarantees that the server is not overloaded. That means that new clients are only admitted if the QoS granted to other clients is not violated. *Statistical admission control* gives such guarantees only with a certain probability [Den95]. Since we assume the presence of video server admission control we only have to regard of the problem of asynchrony from the time a frame is scheduled.

4.2 Sources of Asynchrony in the Server Array

Several sources of asynchrony can be derived for the underlying configuration described in the previous section. Corresponding to section 3.1.1, we can identify the following causes of asynchrony a synchronization mechanism for the Video Server Array has to deal with:

- **Different delays**: the assumption of independent network connections imposes different delays. A synchronization scheme has to compensate for these differences in order to display the continuous media stream in a timely order.
  Beside the network delay, frames experience a delay for the reasons of *packetizing/ depacketizing*, the processing through the lower protocol layers, and the buffering on the client site. Figure 5 shows the cumulative delay of two independent substreams that are started at the same time. The variation of those delays is defined as *jitter.*
- **Network jitter**: asynchronous transfer destroys synchrony. Jitter arises in intermediate nodes for the reason of buffering.
- **End-system jitter**: packetizing and depacketizing of frames with different size due to encoding introduces jitter as well as passing frames through the lower protocol layers.
- **Clock drift** is present because we do not assume global clocks.

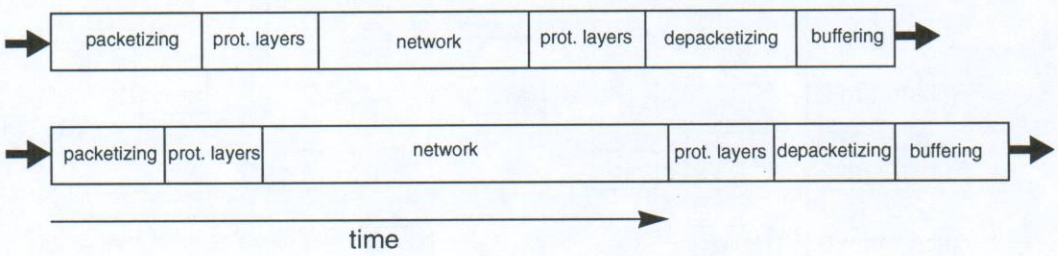

Figure 5: Different Delays Experienced by Independent Substreams.

- **Alteration of the average delay**: the synchronization scheme has to be adaptive with respect to a change of the average delay.[5]
- **Server drop outs** due to process scheduling are a realistic assumption when using non-real-time operating systems. At the same time, the consideration of drop outs covers the overload probability of statistical admission control strategies to a certain amount.

---

[5] Note that an alteration of the average delay belongs to changing network conditions as described in section 3.1.1. We only consider an alteration of the average delay because it strongly affects synchronization. The result is either buffer overflow or buffer starvation.

4.3 Characterization of the Synchronization Problem

Since we assume the video server to deliver only continuous media this clearly creates a *stream synchronization* problem. In contrast to other architectures, a single continuous media stream (e.g. a video stream) in the Video Server Array consists of an arbitrary number of independent *substreams*. For each substream *intra-stream synchronization* has to be provided. The temporal relationship within a substream is given by *r/n*. An example for *n=3* is depicted in figure 6.

A synchronization group is constituted by the number of substreams that contribute to a stream. All members of the group have to be synchronized to each other. Thus, the problem of *inter-stream synchronization* has to be solved. Referring to [Lit90], the temporal relation between the substreams can be characterized by a *before* or an *after* relationship, respectively. Intervals are defined by the frame rate *r* at which the video/audio stream is recorded. For instance, frame number *i* is expected at the client *1/r* seconds *before* frame number *i*+1. We assume that the *i-th* frame is stored on the *(i mod n)-th* server. So, in contrast to classical synchronization problems, we have first a shifted time order between substreams to be synchronized. Second, the highly distributed architecture already imposes the problem of inter-stream synchronization for the delivery of a single continuous media stream.

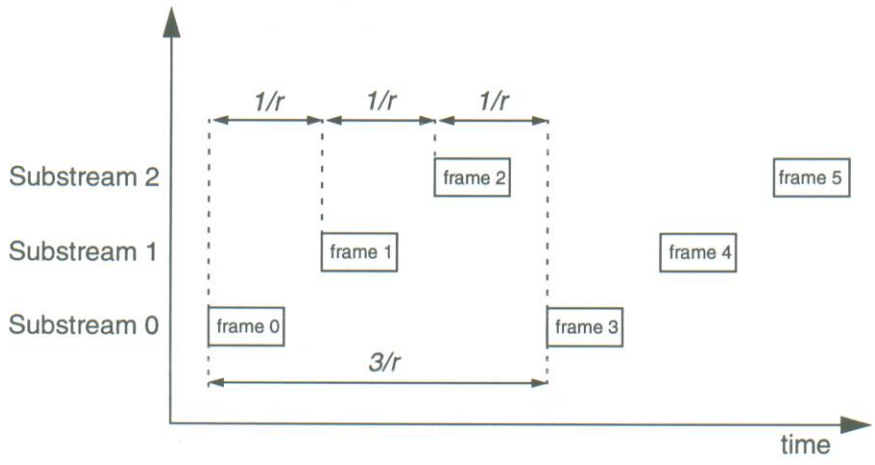

Figure 6: Temporal Relationship for Inter-Frame Striping.

Furthermore, the problem of intra-stream synchronization arises when there exist several synchronization groups to be synchronized to each other. Consider for instance the problem of synchronizing audio and video streams each one consisting of several independent substreams. This kind of synchronization is not subject of the thesis but in section 4.4.7 we give an idea of how the proposed synchronization scheme could be easily extended for the latter problem.

4.4 Synchronization Scheme

4.4.1 Overview

We develop a solution for the given synchronization problem step by step by presenting three models, each one proposing a solution for certain synchronization issues that are described in section 4.2. The models are based on each other and can be characterized by the underlying assumptions. Each one extends its predecessor by dropping assumption taken before. The whole of the models aims at an applicable solution for the considered architecture.

Model 1 covers the problem of *different 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 playback of a media stream that is composed of several substreams in a synchronized manner.

The following second model deals with *jitter* experienced by media units travelling from the source to the sink system. To smoothen out jitter, elastic buffers are required. We present a playout deadline which guarantees a smooth playback of the stream. Based on the deadline buffer requirements are derived for certain scenarios. Model 2 covers intra-stream synchronization as well as inter-stream synchronization. The start-up protocol of model 1 is examined with respect to the effect of jitter. Jitter is assumed to be bounded in this approach.

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 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 selection of model parameters is discussed.

4.4.2 General Assumptions

The synchronization mechanism is based on *time stamps*. Each time a media unit[6] is scheduled by a server, it is provided with the present local time. This enables the client to calculate statistics, e.g. for the roundtrip delay, jitter or inter-arrival times. Moreover, we assume that each frame carries a *sequence number* determining its timely order.

---

[6] For further consideration we will use the terms media unit and frame as synonyms.

In contrast to other approaches buffer requirements or fill levels are always stated in terms of frames or time, instead of the amount of allocated memory. This consideration is preferred because synchronization is a problem of time and for continuous media, time is represented implicitly by the frames of a stream. This seems reasonable because frame sizes vary due to encoding algorithms like JPEG or MPEG [Koe94]. However, notice that a mapping of frames to the allocation of bytes must be carried out for implementation reasons. Taking the largest frame 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 considerations we will use the term *buffer slot* to denote the buffer space for one frame.
Since processing time, e.g. for protocol actions does not concern the actual synchronization problem we will neglect it whereas an implementation has to take it into account. Finally, we assume that control messages are reliably transferred.

4.4.3 Model 1: Start-Up Synchronization

The major problem addressed by model 1 is the compensation for different delays due to the independency of the different substreams. For instance, the geographical distance from server to client may be different for each substream. Thus, starting transmission of frames in a synchronized order would lead to different arrival times at the client with the result of asynchrony. Usually, this is compensated by delaying frames at the client site [Esc94]. Depending on the location of the sources enormous buffering is required.
In order to avoid buffering with regard to the equalization of different delays, we take advantage of the fact that stored media offers more flexibility concerning the temporal handling. The idea is to initiate playout at the servers such that frames arrive on sink site 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. Thus, buffers are replaced by time.
The proposed start-up protocol consists of two phases. In the first phase, roundtrip delays for each substream are calculated while in the second phase the starting time for each server is calculated and propagated back to the servers.
The model is based on the assumption of a constant end-to-end delay without any *jitter.* We further exclude *changing network conditions*, *server drop-outs,* and *clock drift.* In such a scenario synchronization has to be done once at the beginning and is maintained automatically.
The following section introduces a notation which is extended to the subsequent models; interdependencies between the parameters of the model are defined. Afterwards we give a description of the start-up protocol flow and prove its correctness. A generalization of the results follows. We finally close with an example for the protocol.

4.4.3.1 Model Parameters

n	number of server nodes in the server array	
N	number of frames of a stream	
$i,j,\ \upsilon$	frame index	$i, j, \upsilon = 0, ..., N\text{-}1$
k	server index	$k = 0, ..., n\text{-}1$
I_j	index set of n subsequent frames starting with frame j	
S_k	denotes server node k providing substream k	
D	denotes the destination or client node	
r	requested display rate of the entire stream at client site	[fps]
s_i	initial sending time of frame$i$ in server time	[sec]
s_i^c	synchronized sending time of frame i in server time	[sec]
a_i	arrival time of frame i in client time	[sec]
d_i	roundtrip delay[7] for frame i measured at client site	[sec]
d^{max}	maximum roundtrip delay	[sec]
$d^{max,j}$	maximum roundtrip delay for all j element of I_j	[sec]
t_{start}	starting time of the synchronization protocol	[sec]
t_{ref}	reference time for the start-up calculation	[sec]
t_{ref}^j	reference time regarding the set of frames given by I_j	[sec]
t_i	expected arrival of the frame i at the client site	[sec]
δ_{ij}	arrival time difference between frame i and j	[sec]

We already mentioned in section 4.1 that a media stream is assumed to be distributed in a round robin fashion across the involved server nodes. Hence, we can identify the storage location of a frame by its frame number, i.e.

$$\text{Server } S_{i \bmod n} \text{ stores the frame } i. \tag{1}$$

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

$$t_{start} = 0 \tag{2}$$

The index set I_j contains the sequence numbers of n successive frames to be synchro-

[7] The roundtrip delay comprises the delay for a control message that requests a frame and the delay for delivering the frame.

nized by the start-up protocol.

$$I_j = \{j, j+1, \ldots, j+n-1\} \qquad \forall \; j \leq N-n \tag{3}$$

To begin with we regard the first n frames of a stream given by I_0. The roundtrip delay d_i for the frame i is given by the difference between its arrival time a_i and the starting time of the synchronization protocol. Equation (5) states the maximum of the roundtrip delay values.

$$d_i = a_i - t_{start} \quad \forall i \in I_0 \tag{4}$$

$$d^{max} = max\{d_i | i \in I_0\} \tag{5}$$

The second phase of the protocol is launched at time t_{ref} which defines the beginning of the second phase. t_{ref} is determined by the last of the first n frames that arrives.

$$t_{ref} = max\{a_i | i \in I_0\} \tag{6}$$

The presentation rate r determines the arrival time t_i of frame i. After the first frame arrived at t_0 frame i is expected at time t_i given by

$$t_i = t_0 + i \cdot r^{-1} \quad \forall i \tag{7}$$

The difference between the arrival times of arbitrary frames i and j is needed to calculate the starting times of the servers. We define the difference as follows.

$$\delta_{ij} = a_i - a_j \quad \forall i, j \tag{8}$$

4.4.3.2 Start-Up Protocol

The synchronization protocol for starting playback on the server sites is launched after all involved parties are ready for playback. It can be divided into two phases: *evaluation phase* and *synchronization phase*. The goal of the first phase is to compute the roundtrip delays $d_i \quad \forall i \in I_0$ for each connection, while in the second phase starting times are calculated and propagated back to the servers. During the protocol flow the client sends two different kinds of control messages to the servers:

- *Eval_Request(i)*: Client D requests frame i from Server S_i.
- *Sync_Request(i, s_i^c)*: Client D transmits the starting time s_i^c to server S_i.

We now give an overview of the protocol flow while an algorithmic description can be found in appendix A.

(a) Evaluation Phase

- At local time t_{start}, client D sends an *Eval_Request(i)* to Server 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 frame i time-stamped with s_i immediately back to client D, $\forall i \in I_0$.
- At local time a_i, client D receives frame i from Server S_i, $\forall i \in I_0$.
- At local time t_{ref}, client D has received the last frame. The roundtrip delay $d_i = a_i - t_{start} \quad \forall i \in I_0$ and the relative distance between frame arrivals $\delta_{ij} = a_i - a_j \quad \forall i, j \in I_0$ are computed.

(b) Synchronization Phase

- At local time t_{ref}, client D computes $t_0 = max\{t_{ref} + d_i - i \cdot r^{-1} \mid i \in I_0\}$ and the index ν that determines t_0 $\upsilon = \{j \in I_0 \mid t_{ref} + d_j - j \cdot r^{-1} = t_0\}$
- With these results the starting time of Server S_i is calculated in server time $s_i^c = s_i + d^{max} + \delta_{\upsilon i} + (i - \nu) \cdot r^{-1}, \qquad \forall i \in I_o$.
- 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 frame i, $\forall i \in I_0$.
- At local time t_i, client D receives frame i, $\forall i$.

At any time only one frame has to be buffered at the client; after the complete reception the frame played out immediately. The *buffer slot* is now ready to receive the subsequent frame. To show the correctness of our mechanism we cover the two following issues.

- Calculation of t_0
- Calculation of s_i^c

(a) Calculation of the Earliest Possible Playout Time t_0 for the First Frame

We need to choose t_0 such that all frames $i \in I_0$ can be delivered and played out in time, i.e they will arrive at their deadline t_i given by (7). It is obvious that frame $i \in I_0$ delivered by server S_i can not be expected earlier than $t_{ref} + d_i$. Hence, the substream with the largest delay has a significant influence on t_0. In consideration of the distance of $1/r$ between subsequent frames we constitute the following theorem.

Theorem 1: Let $t_0 = max\{t_{ref} + d_i - i \cdot r^{-1} | i \in I_0\}$.
Then all frames can be delivered and played out in time. (9)

Proof: Since the earliest possible arrival time for frame $i \in I_0$ is $t_{ref} + d_i$ we need to show that $t_i \geq t_{ref} + d_i \quad \forall i \in I_0$.
Let $t_0 = max\{t_{ref} + d_i - i \cdot r^{-1} | i \in I_0\}$
$\Rightarrow t_0 \geq t_{ref} + d_i - i \cdot r^{-1} \quad \forall i \in I_0$
$(7) \Rightarrow t_0 = t_i - i \cdot r^{-1} \geq t_{ref} + d_i - i \cdot r^{-1} \quad \forall i \in I_0$
$\Rightarrow t_i \geq t_{ref} + d_i \quad \forall i \in I_0$
$\Rightarrow t_0$ does not violate the arrival times of other substreams
To show that t_0 is minimal we assume that $\exists \ \tilde{t}_0 < t_0$
$\Rightarrow \exists \ i_0 \in I_0$ with $\tilde{t}_0 < t_{ref} + d_{i_0} - i_0 \cdot r^{-1}$
$(7) \Rightarrow \tilde{t}_0 = t_{i_0} - i_0 \cdot r^{-1} < t_{ref} + d_{i_0} - i_0 \cdot r^{-1}$
$\Rightarrow t_{i_0} < t_{ref} + d_{i_0}$ in contradiction to the earliest possible arrival time. ■

For the calculation of the future starting times s_i^c, $\forall i \in I_0$ we define the index determining t_0 as follows.

$$\upsilon = \{j \in I | t_{ref} + d_j - j \cdot r^{-1} = t_0\} \tag{10}$$

(b) Calculation of the Synchronized Sending Time s_i^c of Frame i for Server S_i

Frame υ can be considered critical since it determines the starting times of all other initial frames. It is considered as a reference point to which all other substreams are adjusted. Clearly, the future starting time s_i^c of substream i is composed of the initial starting time s_i plus the maximum roundtrip delay d^{max}. This sum is corrected by the relative arrival time distance $\delta_{\upsilon i}$ between frame i and frame υ. We now have calculated a starting time that provides a simultaneous arrival of the frames of substream i and substream υ. To provide the temporal relationship required by inter-frame striping a multiple of r^{-1} has to be added. The calculation based on (8), (7), (4) is stated in the following theorem.

Theorem 2: Let $s_i^c = s_i + d^{max} + \delta_{\upsilon i} + (i - \nu) \cdot r^{-1}, \forall \ i \in I_o$.
Then frame $i \in I_0$ will arrive at client time t_i. (11)

Proof: For each $i \in I_0$: At client time t_{ref}, s_i^c is sent back to server S_i which receives it at server time $s_i + d^{max}$ (see figure 7). If S_i sent frame i immediately back to D it would arrive at client time $t_{ref} + d_i$. The term $s_i + d^{max}$ is corrected by $\delta_{\upsilon i} + (i - \nu) \cdot r^{-1}$. Thus, frame i will arrive at D at client time

$$
\begin{aligned}
& t_{ref} + d_i + \delta_{\upsilon i} + (i - \nu) \cdot r^{-1} \\
&= t_{ref} + (a_i - t_{start}) + (a_\upsilon - a_i) + (i - \nu) \cdot r^{-1} \\
(2) &= t_{ref} + a_\upsilon + (i - \nu) \cdot r^{-1} \\
&= t_{ref} + d_\upsilon - \upsilon \cdot r^{-1} + i \cdot r^{-1} \\
&= t_0 + i \cdot r^{-1} \\
&= t_i
\end{aligned}
$$

■

4.4.3.3 Generalization

The described scheme can be generalized to any series of subsequent frames requested by the client. One can easily imagine situations where synchronization is needed not only at the beginning of a stream. A typical example is the VCR function *pause*. After having paused it becomes necessary to resynchronize again, starting with the frame subsequent to the last one displayed. We already defined I_j in (3) as a set of subsequent frames starting with j. With little modifications to (4), (5) and (6) we can generalize the protocol. Let j be the first frame of a sequence to be synchronized defined by I_j.

$$d^{max,j} = max\{d_i | i \in I_j\} \tag{12}$$

$$t_{ref}^{j} = max\{a_i | i \in I_j\} \tag{13}$$

Furthermore, the following modification have to be made to (9) and (10).[8]

$$t_j = max\{t_{ref}^{j} + d_i - i \cdot r^{-1} | i \in I_j\} \tag{14}$$

$$\upsilon = \{i \in I_j | t_{ref}^{j} + d_i - i \cdot r^{-1} = t_j\} \tag{15}$$

With (14) and (15) we can compute the starting times s_i^c, $\forall i \in I_j$ analogous to (11).[9]

---

[8] The proof for (14) is analogous to (9).

[9] The proof for (16) is analogous to (11).

$$s_i^c = s_i + d^{max,j} + \delta_{\upsilon i} + (i - \nu) \cdot r^{-1}, \qquad \forall i \in I_j \tag{16}$$

4.4.3.4 Example of the Start-Up Protocol

Model 1 is explained by the following example so to gain a better understanding of the protocol flow. Let $n = 3$ and $r^{-1} = 2$ time units. The calculated starting values are shown in table 1. A time diagram in figure 7 illustrates the protocol flow for the first three frames.

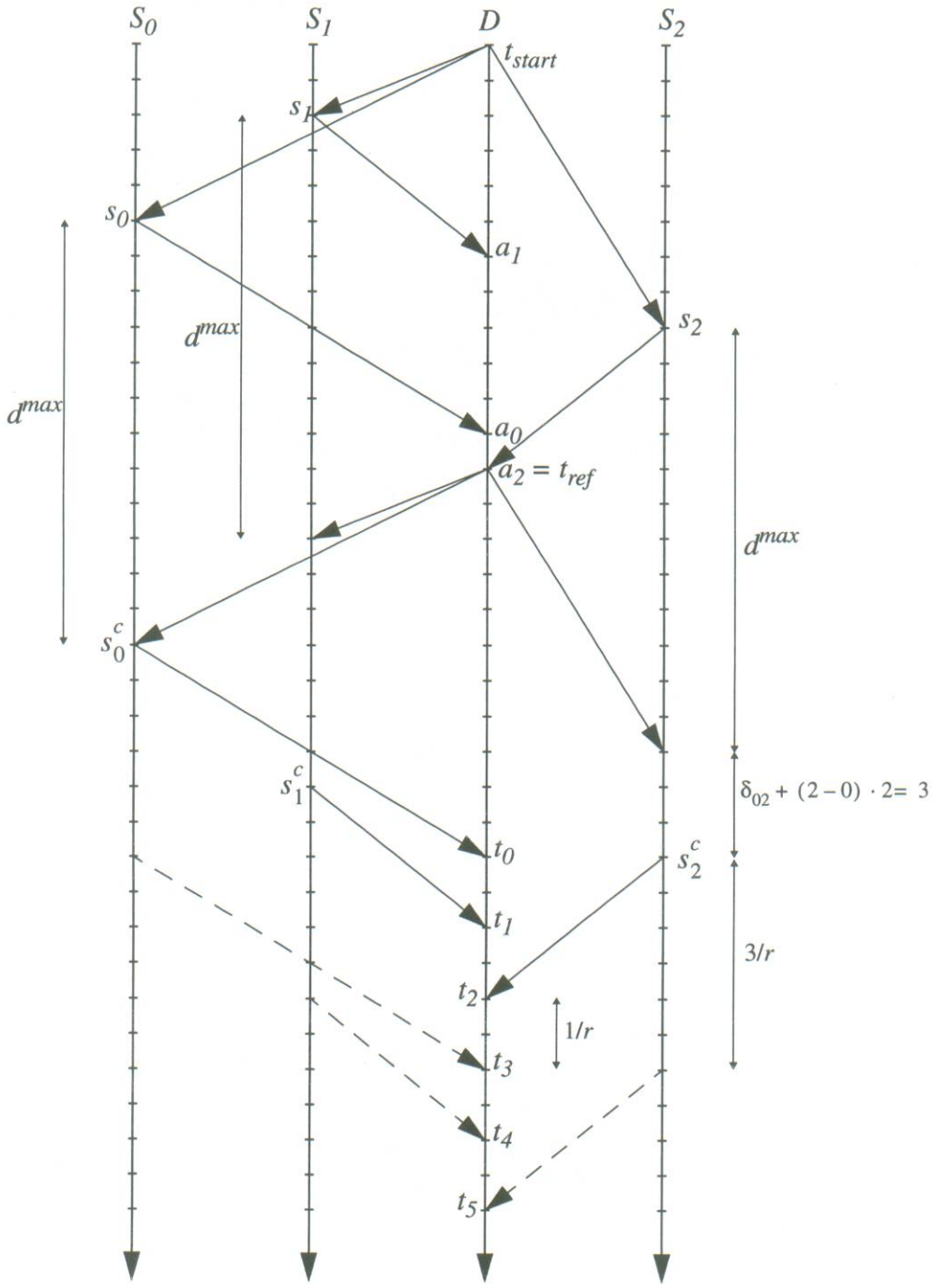

Figure 7: Example of the Start-Up Synchronization Protocol Flow.

For each server and for the client D a time axis is provided. Arrows indicate control messages or frames, respectively, that are transferred between client and servers. Servers always transmit frames. The dotted arrows denote subsequent frames sent by the servers after having been synchronized.

With $t_0 = max\{t_{ref} + d_i - i \cdot r^{-1} \mid i \in I_0\} = max\{23, 16, 20\} = 23$ we get $\upsilon = 0$.

Server	a_i	d_i	t_{ref}	δ_{0i}	s_i^c
S_0	11	11	12	0	$s_0 + 12$
S_1	6	6	12	5	$s_1 + 19$
S_2	12	12	12	-1	$s_2 + 15$

Table 1: Example for the Start-Up Calculations.

Substream 2 experiences the longest roundtrip delay d^{max} and determines therefore t_{ref}. Substream 0 shows to be critical because it cannot be started earlier than $s_0 + 12$. As indicated on the time axis for server 2, substreams 2 could be started earlier but is adjusted to substream 0 as well as substream 1.

4.4.4 Model 2: Intra- and Inter-Stream Synchronization

Model 1 solves the problem of compensating for different delays for each substream. However, synchronization is performed under the assumption that jitter does not exist. Model 2 loosens this assumption and deals with the problem of *end-system jitter* and *network jitter*. For our considerations we regard an accumulated value of all causes of jitter described in section 4.2. Furthermore, we assume that the jitter is bounded.

For the reason of jitter, frames will not arrive in a synchronized manner although they have been sent in a correct timely order. The temporal relationship within one substream is destroyed and time gaps between arriving frames vary according to the occurred jitter. Thus, an isochronous playback cannot be achieved when arriving frames of a substream would be played out immediately. Furthermore, jitter effects lead to a shifting between frames of related substreams in a synchronization group. Hence, *intra-stream synchronization* as well as *inter-stream synchronization* is disturbed. To smoothen out the effects of jitter, frames have to be delayed at the sink such that a continuous playback can be guaranteed. Consequently *playout buffers* corresponding to the amount of jitter are required.

The main point addressed by model 2 is the calculation of the required buffer space.

First, we regard the synchronization of a single substream. Based on a rule of Santoso [San93] we constitute a theorem that states a well defined playout time for a substream such that intra-stream synchronization can be guaranteed. With this so-called *playout deadline* we derive the required buffer space. Smooth playout cannot be guaranteed if starting earlier. 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 [Ish95]. Each one of the substreams is assumed to have a different jitter bound. In this case, buffer reservation according to a single substream is not sufficient anymore as inter-stream synchronization will be disturbed for the reason of differences in the jitter bounds. Additional buffering is required to compensate for this. Furthermore, the playout deadline is modified with respect to multiple substreams.
Finally, we examine the effects of the start-up protocol (model 1) on buffer requirements in the case of jitter. The application of model 1 to initiate playback of the servers in a synchronized manner can introduce an error for the reason of jitter. We give a worst case estimate for the error and additional buffer requirements are computed accordingly.
We begin with an extension of the model parameters used so far.

4.4.4.1 Model Parameters

m	substream/server index,	$m = 0, \ldots, n\text{-}1$
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]
b_k	buffer requirement for substream k on sink site	[frames]
b_k^S	buffer requirement for substream k on sink site with shifting	[frames]
b_k^M	buffer requirement for substream k on sink site with max. jitter	[frames]
B	total buffer requirement for a synchronization group	[frames]
B^S	total buffer requirement for a synchronization groupwith shifting	[frames]
B^M	total buffer requirement for a synchronization group with max. jitter	[frames]

Jitter is usually defined as the variation of network delay. Throughout this section we only regard bounded jitter and we therefore adopt to the definition of jitter given by Rangan et al. [Ran92] who describe jitter as the difference between the maximum delay and the minimum delay.

$$\Delta_k = d_k^{max} - d_k^{min} \quad \forall k \tag{17}$$

$$\Delta^{max} = max\{\Delta_k | k \in \{0 \ldots N-1\}\} \tag{18}$$

In addition to this, we need a jitter bounds defined as the deviation from the average delay $\bar{d}_k$. Jitter is in general not distributed symmetrically. Thus, Δ_k^+ and Δ_k^- must not be equal. For further considerations, we assume interdependencies as follows.

$$\Delta_k = \Delta_k^+ + \Delta_k^- \quad \forall k \tag{19}$$

$$d_k^{max} = \bar{d}_k + \Delta_k^+ \quad \forall k \tag{20}$$

$$d_k^{min} = \bar{d}_k - \Delta_k^- \quad \forall k \tag{21}$$

$$\Delta^{max+} = max\{\Delta_k^+ | k \in \{0 \ldots N-1\}\} \tag{22}$$

4.4.4.2 Synchronized Playout for a Single Substream

To guarantee the timely presentation of a single stream subject to jitter, it is necessary to delay arriving frames at the sink. The buffer compensates for the variable delays. In general, the buffer is emptied at a constant rate for displaying the frames. This strategy is similar to the "leaky bucket" concept employed for instance in ATM networks for the reason of source policing.

Santoso [San93] has already shown that the temporal relationship within one continuous media stream can be preserved by delaying the output of the first frame for $d_k^{max} - d_k^{min}$ seconds. Based on this theorem, both the playout deadline and the buffer requirements can be derived.
The deadline given by the theorem can be lowered in some cases. In the following theorem we state a composed rule for the playout deadline for a single substream.

Theorem 3: Smooth playout for a substream denoted by k can be guaranteed in case of bounded jitter whenever 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 frame

(b) The $(\lceil \Delta_k \cdot r \rceil + 1)$*-th* frame arrived (23)

Proof: A proof for (a) can be found in [San93]. Rule (b) improves (a) in some cases, i.e. playout can start earlier without violating timeliness. Such a situation is shown in figure 9: the first frame experiences the maximum delay, subsequent frames arrive in a burst (marked gray in figure 9) such that after the arrival of the $(\lceil \Delta_k \cdot r \rceil + 1)$*-th* frame the elapsed time is less than $d_k^{max} - d_k^{min}$. The average delay of frames is denoted by dotted lines.

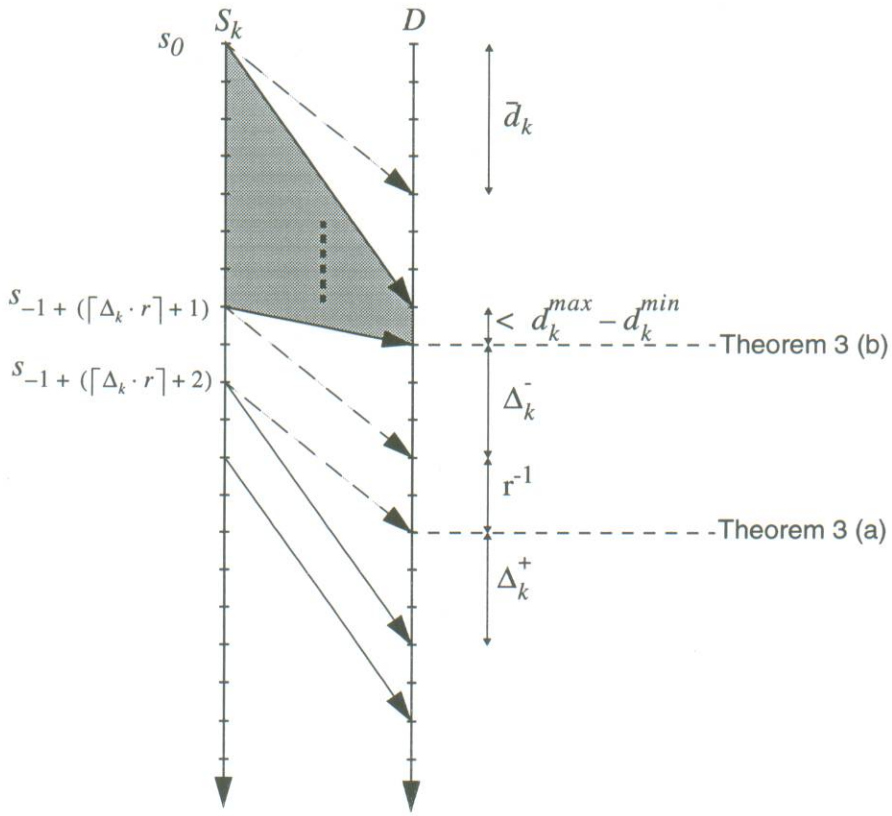

Figure 8: Worst Case Scenario for a Single Substream.

Assuming that the $(\lceil \Delta_k \cdot r \rceil + 1)$*-th* frame just has arrived, we start the playout of the buffered frames immediately. A number of $\lceil \Delta_k \cdot r \rceil + 1$ frames is at least sufficient for a presentation period of $(\lceil \Delta_k \cdot r \rceil + 1) \cdot r^{-1} \geq \Delta_k + r^{-1}$ seconds. In the worst case, the $(\lceil \Delta_k \cdot r \rceil + 1)$*-th* frame experiences its minimum delay and the subsequent frame its

maximum delay. Then the maximum period without any arrival is given by is $\Delta_k^- + r^{-1} + \Delta_k^+ = \Delta_k + r^{-1}$ seconds. $(\lceil \Delta_k \cdot r \rceil + 1) \cdot r^{-1}$ gives an upper bound for $\Delta_k + r^{-1}$. Consequently, the next frame arrives just in time. Following frames will not arrive later because the last one has already experienced the largest delay. ■

Theorem 3 enables us to calculate the required minimum buffer space for the synchronization of a single substream.

Theorem 4: To guarantee intra-stream synchronization for a single substream while applying theorem 3, a minimum buffer space of $\lceil 2\Delta_k \cdot r \rceil$ frames is required. (24)

Proof: To begin with we regard the rule of Santoso stated in theorem 3 (a). We derive the required buffer space based on a worst case scenario outlined in figure 9.

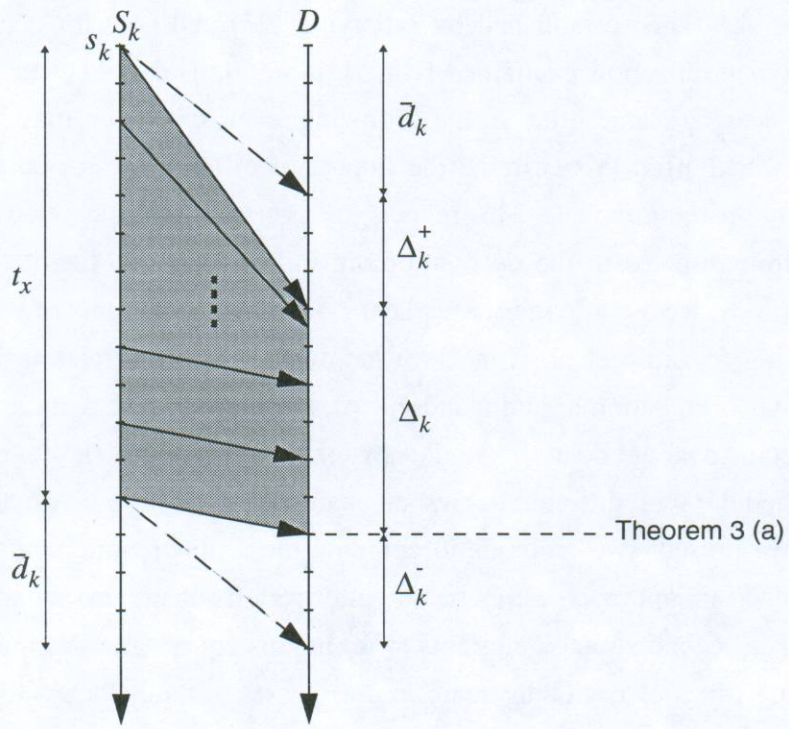

Figure 9: Worst Case Scenario for a Burst Arrival.

Let the first frame of substream k arrive with his maximum delay while all subsequent frames experience their minimum delay. Assuming that no frame can overtake another, in this worst case a burst of frames occurs before the deadline of Δ_k given by theorem 3 (a) is reached. The situation is marked grey in figure 9. The time t_x elapsed between the first frame of the burst and the last one that just arrives at the beginning of the playout

computes as follows:
$t_x = \bar{d}_k + \Delta_k^+ + \Delta_k + \Delta_k^- - \bar{d}_k = 2\Delta_k$. Thus, in worst case the client has to buffer $2\Delta_k$ seconds of playback time, corresponding to $\lceil 2\Delta_k \cdot r \rceil$ frames. Further buffering is not needed because subsequent frames can no arrive earlier.
Theorem 3 (b) improves the rule of Santoso as indicated in the proof for theorem 3. Clearly, playout can start before Δ_k seconds have elapsed as shown in figure 9. But in such a situation the rule of Santoso is equally applicable resulting in a later beginning of the playout, i.e. the consumption from the buffer would start later. We already covered the worst case for theorem 3 (a) and therefore we conclude that by employing theorem 3 (b) no further buffering beyond $2\Delta_k$ seconds of playback time is required. ■

4.4.4.3 Synchronized Playout for Multiple Substreams

The basic idea of the synchronization scheme in model 2 is to achieve inter-stream synchronization between multiple substreams by intra-stream synchronization. Once the latter has been established by satisfying (23) and (24) for each substream, inter-stream synchronization is attained [Ish95], [San93]. This holds true if each substream experiences the same jitter. In the following, each substream may have its own, individually sized buffer. We examine the impact of different jitter bounds for each substream on buffer requirements. This reflects the assumption of the video server array that the paths from sources to the destination are independent. To facilitate the following considerations, we assume the same playout time for media units of each substream[10], i.e. referring to Little et al. [Lit91] we apply the temporal relationship *equality* (e.g. the lip-synchronization of audio and video). Furthermore, we assume that frames experience an equal average delay $\bar{d}$ on all substream connections. The following proofs can also carried out with different delays; an example for (27) can be found in the appendix B.
We present two methods to compute the buffer requirements for multiple substreams. The first approach estimates the jitter for all substreams with the maximum jitter value. The second strategy attempts to refine this coarse-grain estimation by shifting the starting times of each substream in correlation to their jitter values in order to save buffer space. We close this section by considering theorem 3 with respect to the temporal relationship between substreams given by inter-frame striping.

(a) Maximum Jitter Strategy

Obviously, playout can only start if theorem 3 is satisfied for *all* substreams. Thus, the playout deadline for a stream given by a *synchronization group* is defined by the latest

[10] This is in contrast to model 1 where a difference of r^{-1} is required between successive frames.

substream that satisfies (23). The situation is complicated by different jitter bounds for the corresponding substreams which lead to different playout deadlines and buffer requirements. We must avoid a situation where substreams with large jitter bounds still wait for their deadlines while the buffer of other substreams with small jitter bounds already overflows. To cope with this problem in a straight forward manner, Ishibashi et al. [Ish95] propose to allocate the buffer according to the substream with the largest jitter bound. Hence, the buffer requirement b_k^M for each substream of the group and B^M for the complete group are given as follows.

$$b_k^M = \lceil 2\Delta^{max} \cdot r \rceil \tag{25}$$

$$B^M = \sum_{k=1}^{n} b_k^M = n \cdot \lceil 2\Delta^{max} \cdot r \rceil \tag{26}$$

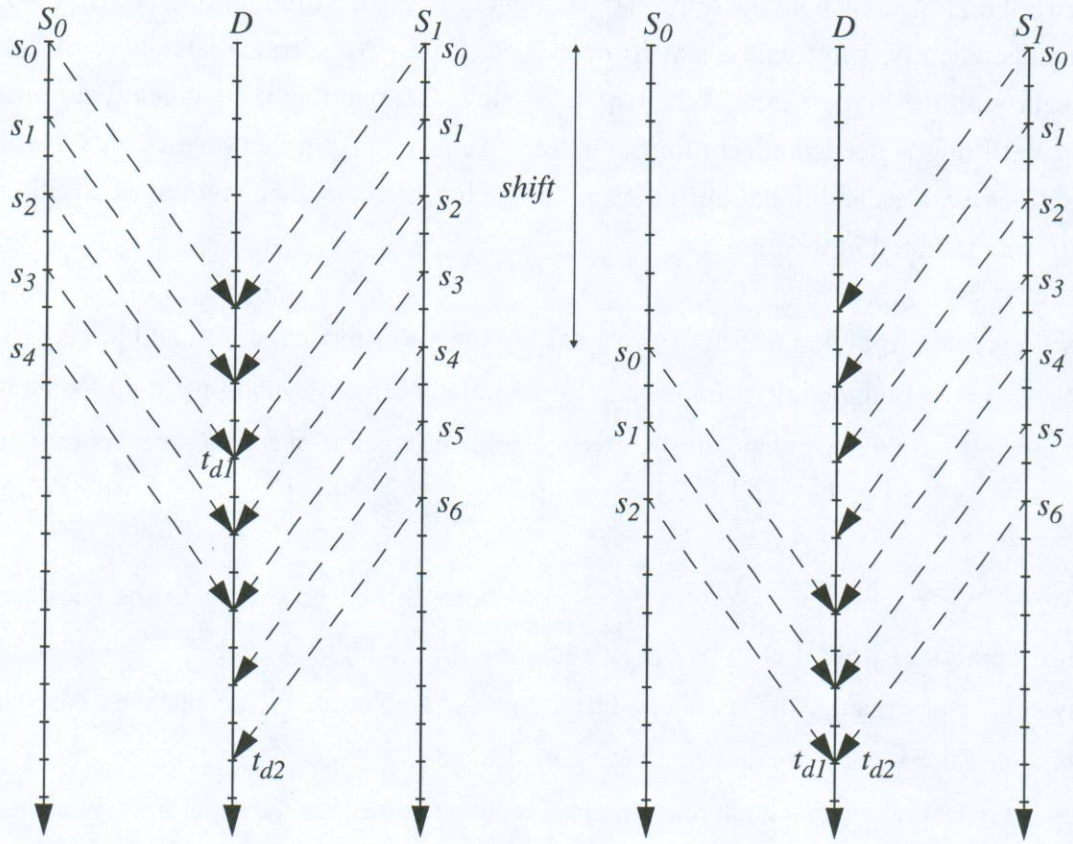

Figure 10: Multiple Substreams without and with Shifting.

(b) Shifting Strategy

Depending on the jitter differences, the maximum jitter strategy might lead to a buffer waste. A more sophisticated way to handle this problem is to synchronize the different substreams such that they reach their playout deadline on average at the same time. This is done by shifting the starting points of all substreams according to the deadline of the substream, with the largest jitter bound. Figure 10 depicts such a scenario for two sources[11] where $\bar{d} = 3.5$, $\Delta_0 = 2$ and $\Delta_1 = 6$.

With (24) we get buffer requirements of four frames for substream 1 and twelve frames for substream 2. Substream 1 reaches its playout deadline on average at t_{d1} and substream 2 at t_{d2}. Without shifting a buffer overflow occurs when receiving the 5th frame of substream 1 while substream 2 still has to wait two time units until playout can commence. With shifting both substreams are due at the same time on average. The amount of the forward shift can be easily derived from (23). The *k-th* substream has to be shifted forward on time axis with the difference of its jitter to the maximum jitter, i.e. $\Delta^{max} - \Delta_k$ seconds. Clearly, substream k has to be started $\Delta^{max} - \Delta_k$ seconds later than the substream with the highest jitter. When applying that shift one might conclude that no further buffering is needed except for the buffer given by (24). In fact, there exists a worst case that requires additional buffer space for each substream. The amount of additional buffering is stated in theorem 5.

Theorem 5: Applying a shift of $\Delta^{max} - \Delta_k$ to the *k-th* substream, $\forall k$, and having bounded jitter for each substream, inter-stream synchronization for multiple dependent substreams can be guaranteed if in addition to theorem 4, $\left\lceil \left(\Delta^{max+} - \Delta_k^+ \right) \cdot r \right\rceil$ buffer slots are allocated. (27)

Proof: We assume that all substreams have been shifted according to the substream with the highest jitter. Let m be the substream with the highest jitter Δ^{max} and let k be any other substream that has been shifted $\Delta^{max} - \Delta_k$ seconds. The proof is based on worst case considerations described in figure 12 and 12.

Regard two substreams which reach their playout deadline[12] at the same time on average due to the application of a shifting. The worst case that can happen is that one substream waits still for its playout deadline while the buffer of the other substream overflows. This becomes true if all frames of one substream experience their minimum delay and all

[11] In contrast to the definitions for model 1, each one of the depicted substreams delivers equal frame numbers.

[12] We refer to the playout deadline given in theorem 3 (a).

frames of the other substream experience their maximum delay. We aligned the playout deadlines of the substreams on average. So, the latest possible playout deadline is given by the substream with the largest positive jitter bound Δ^+ if all frames of this substream experience their largest delay. We can therefore distinguish two cases.

1. $\Delta_m^+ \geq \Delta_k^+$ and the first frame of substream m experiences d_m^{max} while all frames of substream k arrive with their minimum delay d_k^{min}.
2. $\Delta_m^+ < \Delta_k^+$ and the first frame of substream k experiences d_k^{max} while all frames of substream m arrive with their minimum delay d_m^{min}.

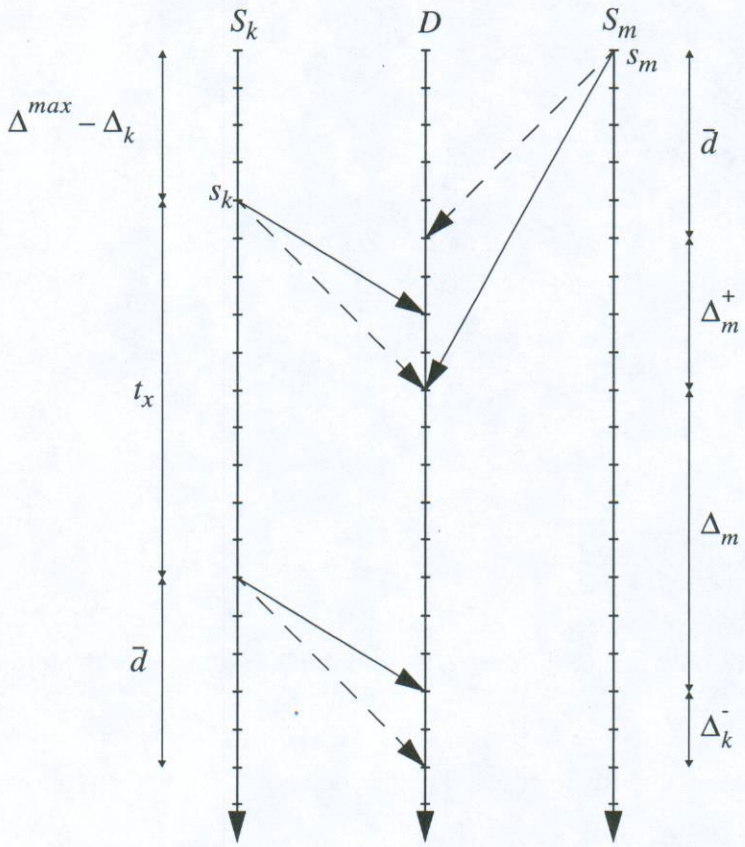

Figure 11: Worst Case Scenario 1 for Multiple Substreams with Shifting

Case 1:

According to theorem 3 the playout deadline for substream m is reached latest $\bar{d} + \Delta_m^+ + \Delta_m$ seconds after the first frame of m has been sent. We are interested in the maximum number of frames of the *k-th* substream that may arrive before the playout deadline is reached. We consider the period t_x between the first frame of the *k-th* substream and that frame of the *k-th* substream that arrives just at the playout deadline of the *m-th* substream.

$$t_x = \bar{d} + \Delta_m^+ + \Delta_m + \Delta_k^- - \left(\bar{d} + \Delta^{max} - \Delta_k \right)$$

$$= \bar{d} + \Delta_m^+ + \Delta_m^+ + \Delta_m^- + \Delta_k^- - \left(\bar{d} + \Delta_m^+ + \Delta_m^- - \Delta_k^+ - \Delta_k^- \right)$$
$$= \Delta_m^+ + 2\Delta_k^- + \Delta_k^+$$

Applying theorem 3, we already allocated $2\Delta_k$ buffer space in terms of time. From this we conclude an additional buffering of $\left\lceil \left(\Delta_m^+ + 2\Delta_k^- + \Delta_k^+ - 2\Delta_k \right) r \right\rceil$
$= \left\lceil \left(\Delta_m^+ - \Delta_k^+ \right) r \right\rceil$ frames for the *k-th* substream.

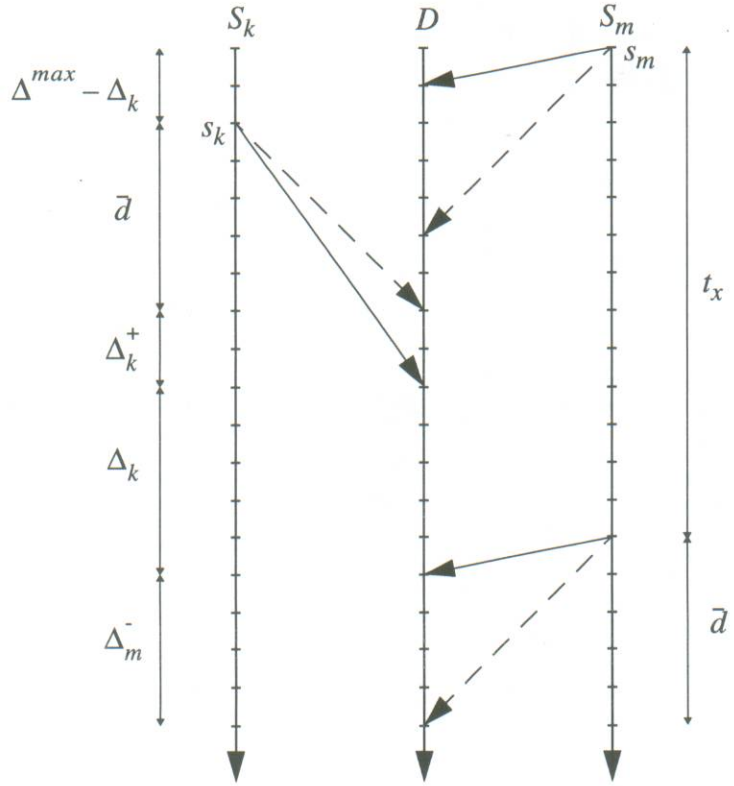

Figure 12: Worst Case Scenario 2 for Multiple Substreams with Shifting.

Case 2:

Case 2 can be shown analogous to case1. The period t_x is computed as follows.

$$t_x = \left(\Delta^{max} - \Delta_k \right) + \bar{d} + \Delta_k^+ + \Delta_k + \Delta_m^- - \bar{d}$$
$$= \left(\Delta_m^+ + \Delta_m^- - \Delta_k^+ - \Delta_k^- \right) + \Delta_k^+ + \Delta_k^+ + \Delta_k^- + \Delta_m^-$$
$$= \Delta_k^+ + \Delta_m^+ + 2\Delta_m^-$$

We already allocated $2\Delta_m$ buffer space for substream *m*. Thus, we get additional buffering of $\left\lceil \left(\Delta_k^+ + \Delta_m^+ + 2\Delta_m^- - 2\Delta_m \right) r \right\rceil = \left\lceil \left(\Delta_k^+ - \Delta_m^+ \right) r \right\rceil$ frames for the *m-th* substream.

We can conclude that the additional required buffer space does not depend on the substream with the highest jitter but on that one with the highest upper jitter bound Δ^{max+} defined in (22). We can therefore derive an additional buffering of $\left\lceil \left(\Delta^{max+} - \Delta_k^+ \right) \cdot r \right\rceil$ frames for an arbitrary substream *k*. ■

In contrast to (25) and (26) the total buffer requirements can be computed by applying theorem 3 and 4 as follows.

$$b_k^S = \left\lceil \left(2 \cdot \Delta_k + \Delta^{max+} - \Delta_k^+ \right) \cdot r \right\rceil \tag{28}$$

$$B^S = \sum_{k=1}^{n} b_k^S = \sum_{k=1}^{n} \left\lceil \left(2 \cdot \Delta_k + \Delta^{max+} - \Delta_k^+ \right) \cdot r \right\rceil \tag{29}$$

Theorem 6: Applying the shifting strategy for the synchronization of multiple substreams saves buffer space independent of values for Δ_k, Δ_k^+ and Δ_k^-, $\forall k$. (30)

Proof: To prove theorem 6 we have to show that $b_k^S \le b_k^M$. For the following considerations we will express the buffer requirements in terms of time. Let

$\hat{b}_k^S = 2 \cdot \Delta_k + \Delta^{max+} - \Delta_k^+$, $\forall k$ and

$\hat{b}_k^M = 2 \cdot \Delta^{max}$, $\forall k$.

Then, $\hat{b}_k^M - \hat{b}_k^S$

$= 2 \cdot \Delta^{max} - \left(2 \cdot \Delta_k + \Delta^{max+} - \Delta_k^+ \right)$

$= 2 \cdot \Delta^{max} - 2\left(\Delta_k^+ + \Delta_k^- \right) - \Delta^{max+} + \Delta_k^+$

$= 2 \cdot \Delta^{max} - \Delta_k^+ - 2 \cdot \Delta_k^- - \Delta^{max+}$

$= 2 \cdot \Delta^{max} - \Delta_k - \Delta_k^- - \Delta^{max+}$

$= 2 \cdot \Delta^{max} - (\Delta_k) - \left(\Delta_k^- + \Delta^{max+} \right) \ge 0$

because $\Delta_k \le \Delta^{max}$ with (18) and $\Delta_k^- + \Delta^{max+} \le \Delta^{max}$ with (19), (22).

$\Rightarrow \hat{b}_k^S \le \hat{b}_k^M$ ■

To demonstrate the buffer savings, we computed $B^M - B^S$ for two substreams. For substream 0 we have chosen a fixed jitter value of $\Delta_0 = 40$ ms while Δ_1 is varied with respect to Δ_0 in steps of 20 ms, taking the values 60, 80, ..., 200 ms. For each substream we admitted three values for Δ_k^+:

- High: $\Delta_k^+ = 3/4 \cdot \Delta_k$
- Medium: $\Delta_k^+ = 1/2 \cdot \Delta_k$
- Low: $\Delta_k^+ = 1/4 \cdot \Delta_k$

We calculated the shift of $\Delta_1 - \Delta_0$ accordingly to theorem 5 and for each of the shift values, we regarded all possible combinations of the Δ_k^+ values between the two substreams.

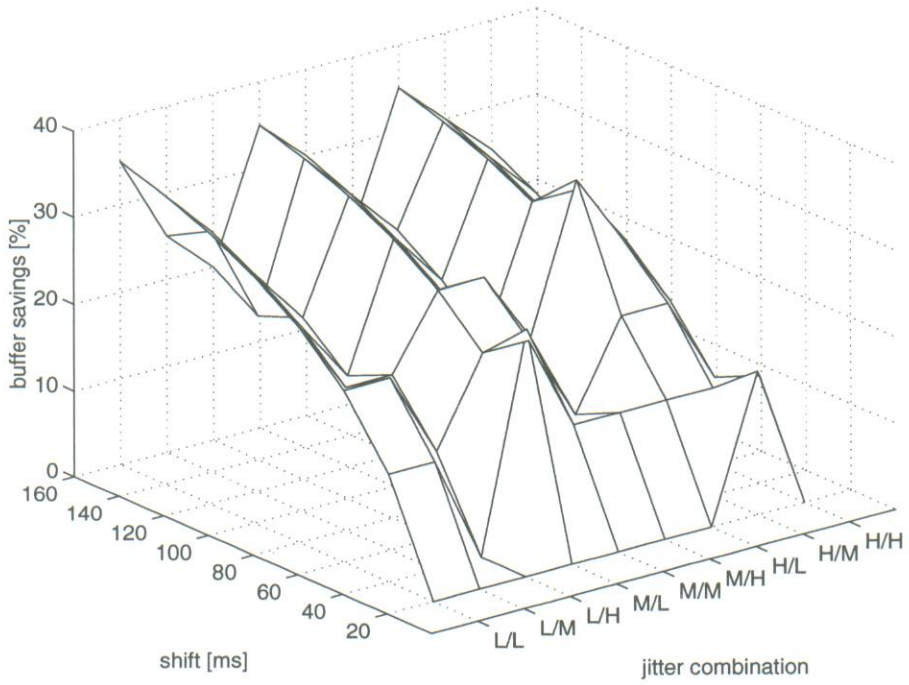

Figure 13: Buffer Saving for Different Shifts and Jitter Combinations.

Δ_0	Δ_1	shift	jitter combination									
			L/L	L/M	L/H	M/L	M/M	L/H	H/L	H/M	H/H	
40	60	20	6	6	6	6	6	6	6	6	6	B^M
			6	6	6	6	6	6	6	5	6	B^S
			0	0	0	0	0	0	0	16.7	0	%
40	80	40	8	8	8	8	8	8	8	8	8	B^M
			7	7	8	6	7	7	7	7	7	B^S
			12.5	12.5	0	25	12.5	12.5	12.5	12.5	12.5	%
40	100	60	10	10	10	10	10	10	10	10	10	B^M
			8	8	9	8	8	9	8	8	9	B^S
			20	20	10	20	20	10	20	20	10	%
40	120	80	12	12	12	12	12	12	12	12	12	B^M
			9	10	10	9	9	10	8	9	10	B^S
			25	16.7	16.7	25	25	16.7	33	25	16.7	%
40	140	100	14	14	14	14	14	14	14	14	14	B^M
			10	11	12	10	11	12	10	11	12	B^S
			28.57	21.43	14.29	28.57	21.43	14.29	28.57	28.57	21.43	%
40	160	120	16	16	16	16	16	16	16	16	16	B^M
			11	12	13	11	12	13	11	12	13	B^S
			31.25	25	18.75	31.25	25	18.75	31.25	25	18.75	%
40	180	140	18	18	18	18	18	18	18	18	18	B^M
			12	13	15	12	13	14	12	13	14	B^S
			33.33	27.78	16.67	33.33	27.78	22.22	33.33	27.78	22.22	%
40	200	160	20	20	20	20	20	20	20	20	20	B^M
			13	15	16	13	14	16	13	14	15	B^S
			35	25	20	35	30	20	35	30	25	%

Table 2: Computed Buffer Savings.

Combinations are denoted by a pair Δ_0^+/Δ_1^+, were Δ_0^+ and Δ_1^+ can take the values H, M, and L for High, Medium, and Low. e.g. (h/m). The values for B^M and B^S shown in table 2 are rounded up based on a frame rate of 25 fps. Figure 13 depicts the buffer savings for the different values and jitter combinations.

For an arbitrary combination of jitter values, buffer savings increase when the shift between the two substreams becomes large, this is, the larger the difference in jitter values between substreams the more buffer is saved due to theorem 5. For a shift of 160 ms for instance, buffer savings up to 35% compared to the maximum jitter strategy are possible. If the difference in jitter between two substreams equals zero, both the maximum jitter strategy and the shifting strategy require an equal buffering.

For the combinations of jitter values, a wave form can be observed in figure 13. The lower the maximum value Δ^{max+} the more buffers are saved (compare proof for (30)). Since Δ_0 is always smaller than Δ_1, Δ^{max+} gets minimal for low Δ_1^+ values. Thus, we obtain the highest buffer savings with the combinations L/L, M/L, and H/L.

(c) Playout Deadline for Inter-frame Striping

So far, we have only regarded the simultaneousness of playout of the different substreams to facilitate our considerations. We will extend this to inter-frame striping where frames of different substreams are played out subsequently. Hence, we have a temporal relationship like *substream* 0 *before substream* 1, or *substream* 1 *after substream* 0, respectively. The relationship within one substream is defined by *r/n*, while the complete stream at the sink site is played out with the rate *r*, i.e. subsequent frame numbers are expected to arrive with a distance of r^{-1}. These temporal needs have already been broached by model 1 and are now examined with respect to the playout deadline.

Applying theorem 3, playout is started when each of the substream satisfies the starting conditions. This guarantees a simultaneous playout for each one of the substreams though it is not needed. Subsequent frames are required to be played out r^{-1} seconds later, relative to their predecessors. Thus, the idea is simply to lower the playout deadlines for subsequent substreams according to their distance to substream 0.

Lemma 1: Smooth playout of a synchronization group in the case of bounded jitter can be guaranteed whenever one of the following starting conditions holds true for each substream *k*.
(a) $d_k^{max} - d_k^{min} - (k-1)\,r^{-1}$ seconds elapsed after the arrival of the first frame of substream *k*.

(b) The $(\lceil \Delta_k \cdot r/n \rceil)$-*th* frame of substream k arrived and since then $(n-k+1) \cdot r^{-1}$ seconds have elapsed. (31)

Proof: The first frame of the first substream is expected to be played out first. Obviously, to guarantee smooth playout substream 0 has to satisfy theorem 3. Let k be an arbitrary substream of a synchronization group. The first frame of k is expected $(k-1)\,r^{-1}$ seconds after the first frame of substream 0. We assume that theorem 3 is satisfied for the first substream and that at least $d_k^{max} - d_k^{min} - (k-1)\,r^{-1}$ seconds have elapsed since the first frame of the substream k has arrived. When the playout of the first frame is started immediately we know exactly that the *k-th* frame is needed to be played out in $(k-1)\,r^{-1}$ seconds. The time passed till this moment amounts at least up to $d_k^{max} - d_k^{min} - (k-1)\,r^{-1} + (k-1)\,r^{-1} = d_k^{max} - d_k^{min}$ seconds.
With this we can conclude that theorem 3 is fulfilled for any substream k when applying lemma 1 (a).

To prove lemma 1 (b), we argue analogous to the proof of theorem 3. Again, we assume that theorem 3 is satisfied for the first substream. Further, the $(\lceil \Delta_k \cdot r/n \rceil)$-*th* frame of substream k has arrived and $(n-k+1) \cdot r^{-1}$ seconds have elapsed since the arrival. Playout of the first frame is started instantaneously. An amount of $\lceil \Delta_k \cdot r/n \rceil$ frames is at least sufficient for a presentation of $\lceil \Delta_k \cdot r/n \rceil \cdot (r/n)^{-1} \geq \Delta_k$ seconds. In the worst case (see (23)) the maximum period between the arrival of the $(\lceil \Delta_k \cdot r/n \rceil)$-*th* frame and the $(\lceil \Delta_k \cdot r/n \rceil + 1)$-*th* frame is $\Delta_k + n/r$ seconds. An period of $(n-k+1) \cdot r^{-1}$ seconds has already elapsed and further, frames of the substream k are expected $(k-1)\,r^{-1}$ seconds later than the first frame of substream 0. We can therefore compute the total elapsed time by
$\left((n-k+1) \cdot r^{-1}\right) + \left((k-1)\,r^{-1}\right) = nr^{-1} - kr^{-1} + r^{-1} + kr^{-1} - r^{-1} = n/r.$
Thus, the worst case is covered and frame $(\lceil \Delta_k \cdot r/n \rceil + 1)$ will arrive in time. We can conclude that theorem 3 is fulfilled for any substream k when applying lemma 1 (b). ■

4.4.4.4 Start-up Protocol Influence

Until now, we have assumed that substreams are synchronized with respect to their average delay by some kind of mechanism, i.e. frames arrive on average in a synchronized manner (see dotted lines in the previous figures). Model 1 is based on jitter-free network connections. Employing the scheme in the case of bounded jitter cannot guarantee the synchronization of the substreams with respect to their average delay. The start-up calcu-

lation is based on the roundtrip delay values experienced by the first n frames. Only if the observed delay corresponds to the average delay, the start-up protocol proves correct. However, the delay can be altered due to jitter, hence the calculation introduces an error. The worst case must be covered by buffering.

The start-up protocol computation is based on a roundtrip delay for a request packet and one or several packets carrying a frame. Actually, jitter arises for transmitting the request packet from sink to source and for sending the frame back to the client.In general, the request message is a small packet made up of several bytes. Hence, in the following considerations we will neglect the jitter experienced by the request packet, supposing that the jitter bounds for the buffer calculations stated in (24) and (27) have been chosen sufficiently large.

Theorem 7: When applying a shift of $\Delta^{max} - \Delta_k$ to the *k-th* substream, $\forall k$ in the case of bounded jitter, and when using the start-up protocol given by model 1, intra-stream and inter-stream synchronization for multiple dependent substreams can be guaranteed if additionally to theorem 4 and 5

$$\left\lceil max\{\Delta_m + \Delta_k^+ - \Delta^{max+} \,|\, m \neq k \wedge m = 1 \ldots n\} \cdot r/n \right\rceil$$

frames are buffered. (32)

Proof: The proof is analogous to (27) except that an additional shift introduced by the start-up calculation is taken into account. Consider two substreams k and m where $m>k$ and $\Delta_m > \Delta_k$. The worst case that can happen during the start-up protocol is that the calculation for substream k is based on its maximum delay but subsequent frames arrive with their minimum delay, and for the other substream m we calculate with its minimum delay but subsequent frames arrive with their maximum delay.[13] The scenario is depicted in 12. The dotted lines denote the delay assumed by the start-up protocol. The occurred error is indicated gray.

Consider now a shift of $(m-k)\,r^{-1}$ seconds that has been introduced by the start-up protocol. Further, we employ lemma 1. Consequently, the playout deadline for substream m is reached latest $\bar{d} + \Delta_m + \Delta_m - (m-k)\,r^{-1}$ seconds after frame m is sent. We are interested in the maximum number of frames of substream k that can arrive before the playout deadline is reached. The period t_x between the first frame of substream k and a frame that arrives just at the playout deadline of substream m computes as follows.

[13] Tis corresponds to the worst case scenario described in the proof for theorem 5.

$t_x = (m-k)\,r^{-1} + \bar{d} + \Delta_m + \Delta_m - (m-k)\,r^{-1} + \Delta_k - (\Delta_m - \Delta_k + \bar{d})$

$= 2\Delta_k + \Delta_m$

We already considered $2\Delta_k + \left(\Delta^{max+} - \Delta_k^+\right)$ seconds in terms of buffering by employing theorem 4 and 5 for substream k. Thus, an additional buffering of

$2\Delta_k + \Delta_m - \left(2\Delta_k + \left(\Delta^{max+} - \Delta_k^+\right)\right)$

$= \Delta_m + \Delta_k^+ - \Delta^{max+}$ seconds of playback time corresponding to

$\left\lceil \left(\Delta_m + \Delta_k^+ - \Delta^{max+}\right) \cdot (r/n) \right\rceil$ frames is required for substream k. We extend this consideration to the whole synchronization group by simply comparing each substream k with all others of the group. We calculate the additional buffer requirements as indicated in the proof and we then take the maximum of these values as the actual additional buffer for substream k, $\forall k$. ■

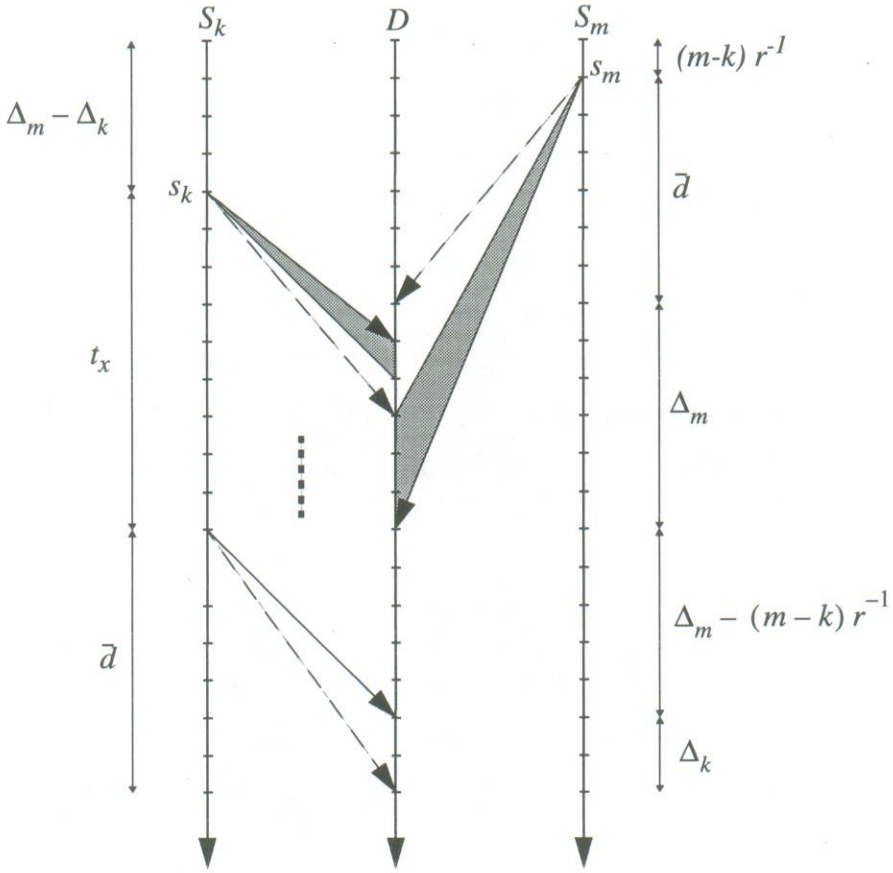

Figure 14: Worst Case Scenario for Start-Up Protocol Influence.

4.4.4.5 Optimization

Model 2 gives us a framework to compute buffer requirements for multiple substreams with different jitter bounds to attain inter-stream synchronization by maintaining intra-stream synchronization. Buffer requirements are given by theorem 4 and 5. The error

introduced by the start-up protocol is corrected by theorem 7, and we have a well-defined playout deadline for all substreams as defined in lemma 1. Throughout all theorems, we expressed the time to buffer in terms of frames. The required buffer space can be optimized by adding the time to buffer given by theorem 4, 5 and 6 and by transforming the resulting sum into buffer slots. Hence, we can summarize the overall buffer requirements b_k for a substream k and B for a synchronization group consisting of n substreams as follows.

$$b_k = \left\lceil \left(2\Delta_k + \left(\Delta^{max+} - \Delta_k^+ \right) + \right. \right. \tag{33}$$
$$\left. \left. max\{\Delta_m + \Delta_k^+ - \Delta^{max+} \middle| m \neq k \wedge m = 1 \ldots n\} \right) \cdot (r/n) \right\rceil$$

$$B = \sum_{k=1}^{n} b_k =$$

$$\sum_{k=1}^{n} \left\lceil \left(2\Delta_k + \Delta^{max+} - \Delta_k^+ + max\{\Delta_m + \Delta_k^+ - \Delta^{max+} \middle| m \neq k \wedge m = 1 \ldots n\} \right) \cdot (r/n) \right\rceil \tag{34}$$

4.4.5 Model 3: Resynchronization

Under the assumption of bounded jitter we can guarantee both intra-stream synchronization and inter-stream synchronization by applying model 1 and 2. Using ATM based networks, this assumptions holds true at least for the network because we can express the acceptable QoS in parameters like throughput, delay, jitter or cell losses [Cor92]. Since we regard accumulated jitter that includes the end-system jitter, we cannot assume to have any boundaries; especially non-real-time operating systems cannot provide any guarantees.

However, in case of unbounded jitter an application is forced to make certain assumptions on jitter as either resources, respectively buffer space, are restricted or the increase in end-to-end delay by buffering is unacceptable [Cor92]. If the statistical distribution of the end-to-end delay is known a priori, jitter bounds can be chosen according to the desired QoS of synchronization. With a given probability α of an error we can state that with a probability $1 - \alpha$ the delay of transmitted frames stays within the chosen bounds. Exceeding the bounds results in buffer starvation or overflow; related frames are discarded. Throughout the following considerations we assume that jitter bounds are selected according to an acceptable quality of synchronization.

Model 3 can be characterized as a scheme for resynchronization. We apply the concept

of a *buffer level control* to detect asynchrony, and to recover from asynchrony we use feedback messages to the servers. According to the description of the actual synchronization problem in section 4.2 model 3 has to cope with

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

An *alteration of the average delay* might leads to a *gap* or a *concentration* in the continuous media stream. A gap occurs when the average delay gets longer, a concentration can be observed when it shortens. The situation is illustrated in figure 15.[14] The playout intervals at the client site are marked on the time axis. Arriving frames are represented by arrows. Notice that an alteration of the average delay is assumed to be of long-term effect, otherwise a disturbance is already covered by the normal buffering for the reason of jitter. A gap or a concentration leads to a shifting of the average buffer level obtained after the substream has been started by employing model 1. Hence, intra-stream synchronization and consequently inter-stream synchronization is disturbed if we assume the jitter bounds to stay constant. Depending on the extent of the shifting a rising number of lost frames up to total buffer starvation/overflow may be observed.

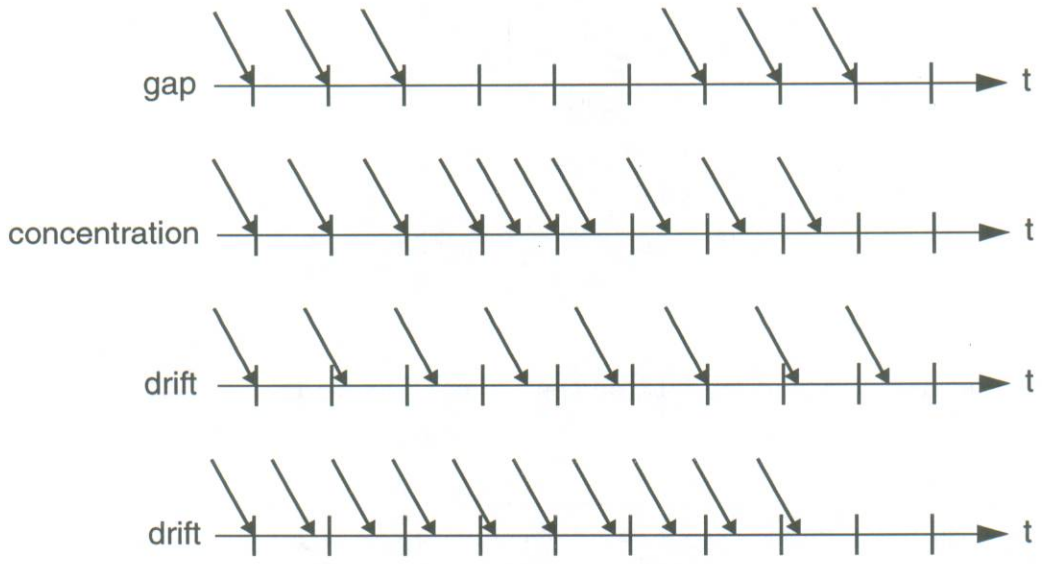

Figure 15: Different Types of Disturbances.

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 as defined in section 3.2. If a server clock is

[14] Note that the frames in the gap are not lost and frames in a concentration are not doubled.

faster than the client clock (determining the consumption rate), the scheduling frequency will be higher on the server site than on the client site. Thus, regarding an arbitrary time interval, the arrival rate is higher than the consumption rate, i.e. the utilization ratio becomes less than 1. This process accumulates and leads to a buffer overflow. In contrast, if the server clock is slower than the clock on the client site, the scheduling frequency will be lower on the server site than on the client site, i.e. the utilization ratio becomes greater than 1, resulting in buffer starvation.

We assume to have some kind of admission control in the underlying server which either stochastically or deterministically guarantees that the server is not overloaded. Nonetheless, it might be possible for reasons specific to the used operating system that a *drop out* occurs during the scheduling of a video, e.g. when a new process or a daemon is launched. The consequence is a gap in the continuous media stream.

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.

We have shown that all the described disturbing factors affect the buffer level. Thus, the buffer level can be regarded as an indicator for upcoming synchronization troubles. Once a sink has discovered a problem it has to take measures, accordingly, in order to restore synchronization. As the reason for asynchrony is basically a shifting in the media stream we only have to correct this shifting. Referring to section 3.3 we already know that resynchronization can be done either by adapting the production or presentation rate, or by skipping/pausing. Corrective actions have to 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 *buffer level control*. Basic work in this area can be found in [Rot95b], [Koe94] and [Lit92]. Our model will pick up some of their basic ideas and extend them to an applicable solution for the synchronization problem. In contrast to their 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.

The next section gives an overview of the used parameters, afterwards models 1 and 2 are examined with respect to a buffer level control, this is we present a buffer model suit-

able to realize a buffer level control. Finally, we discuss the degree of resynchronization action and duration issues.

4.4.5.1 Model Parameters

UW_k	upper buffer watermark for substream k	[frames]
LW_k	lower buffer watermark for substream k	[frames]
b_k^A	additional buffer slots for substream k	[frames]
B_k	total buffer size for substream k	[frames]
q_{tk}	queue size for substream k at time t	[frames]
$\bar{b}_{tk}$	smoothed buffer level substream k at time t	[frames]
o_{tk}	computed resynchronization offset	[frames]
$S(q_{tk})$	smoothing/filtering function	[frames]
$C(\bar{b}_{tk})$	control function	[frames]
$\bar{d}_k^{new}$	new average roundtrip delay	[sec]
R	length of resynchronization phase	[sec]
α	smoothing factor	

4.4.5.2 Buffer Level Control

(a) System Model

The scheme of buffer level control is often referred to as a *control loop* [Koe94]. Sources transfer frames over the network that arrive at the sink site 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. [Koe94], [Rot95b] 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. A similar approach is proposed by Cen et al. [Cen95] who describe the feedback mechanism in a distributed MPEG player. As shown in figure 16, the archi-

tecture applies feedback actions to the sources via control messages in order to maintain synchronization at the sink.

The buffer level for substream k at time t is denoted by q_{tk}. This value is periodically passed to a filtering function $S(q_{tk})$ so to filter short-term fluctuations caused by jitter. Examples for filtering functions are the geometric weighting smoothing function [Rot95b], [Cen95], [Mas90]:

$$S(q_{tk}) = \alpha \cdot \bar{b}_{t-1k} + (1-\alpha) \cdot q_{tk} = \bar{b}_{tk} \quad \text{with } \alpha \in [0,1], \tag{35}$$

or the finite impulse response filter used by Koehler et al. [Koe94]. 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. The tuning of a filter is discussed with (35) in the next section.

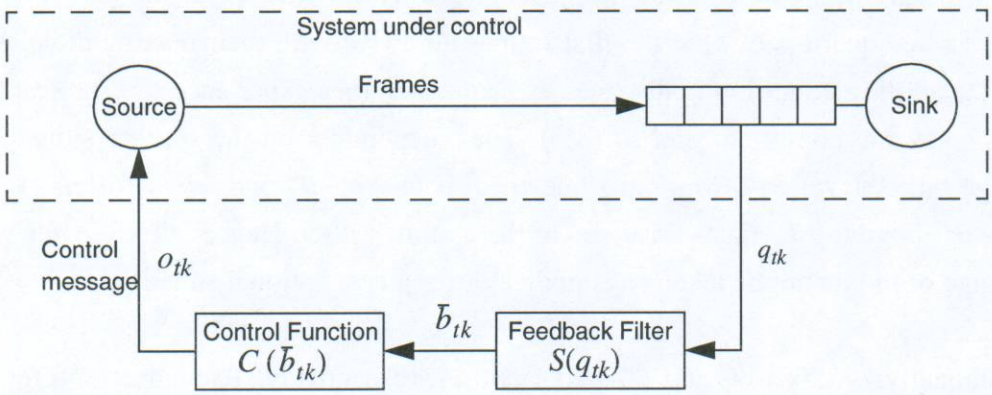

Figure 16: System Model for the Buffer Level Control [Cen95].

The smoothed buffer level $\bar{b}_{tk}$ is passed to a control function $C(\bar{b}_{tk})$ which 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 frames specified in the offset or pauses for a duration of o_{tk} frames. 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. The QoS of other clients serviced by the video 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 has moved back in the normal area into between LW_k and UW_k. If not, a new resynchronization phase is started. [Rot95b]

(b) Buffer Requirements and Filter Tuning

With the estimation or choice of jitter bounds according to a certain QoS for synchronization we can compute the buffer requirements by employing models 1 and 2. The resulting buffer space b_k can be regarded as a so-called *kernel buffer*. Applying a buffer level control only to this buffer is vain since each buffer level within the range of b_k must be regarded as normal due to jitter effects or starting conditions. Especially starting conditions given by theorem 3 and by lemma 1 have a strong impact on the subsequent fill level of the buffer: assume a buffer corresponding to $4\Delta_k$ seconds of playback time, then a substream may be started already with a fill level of one frame if the first frame experiences its minimum delay and the subsequent one its maximum delay. The average buffer level will tend to a quarter of b_k in this situation. Vice versa, we obtain an average fill level of three quarters of b_k if the first frames all arrive with their maximum delay and after the starting condition holds true, subsequent frames experience the shortest delay (for the starting condition refer to (31)). Thus, depending on the starting situation the average buffer level will even out somewhere between $\frac{1}{4}b_k$ and $\frac{3}{4}b_k$. Buffer levels q_{tk} below or above those values arise due to the assumed jitter. Hence, all fill levels within the range of b_k can not be taken as an indicator for an exceptional situation.

Consequently, we fix LW_k and UW_k to 1 and b_k, respectively. To realize a buffer level control we of course have to admit buffer levels below and above the watermarks. Otherwise it is impossible to get the smoothed buffer level below or above the watermarks.

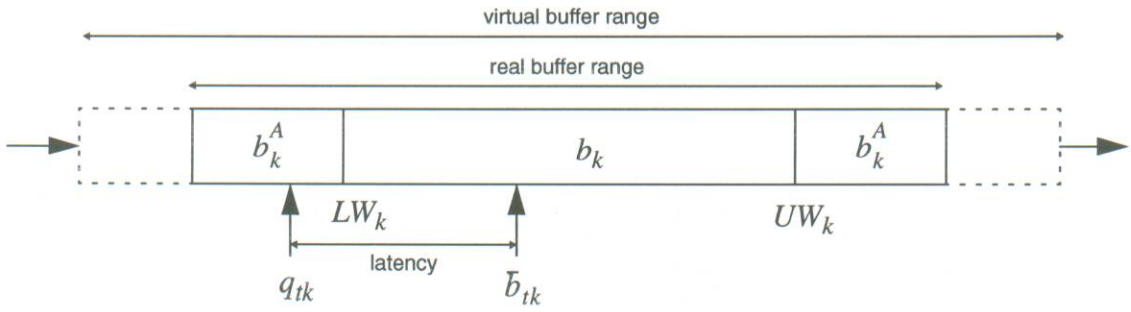

Figure 17: Buffer Model with Virtual and Real Buffer.

We suggest the scheme of a so-called *virtual buffer* as indicated in figure 15 by dotted lines. The virtual buffer includes at least the real buffer given by the kernel buffer b_k and by an additional buffer b_k^A. As shown later, the additional buffer b_k^A above and below the kernel buffer determines the policy of resynchronization. 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.

1. If neither buffer starvation nor buffer overflow occurs, the real buffer level equals the virtual buffer level.
2. If a buffer overflow occurs, then the virtual buffer is increased for each discarded frame while the real buffer levels resides unchanged.
3. If a buffer starvation occurs, then the virtual buffer is decreased each time when the client scheduling finds an empty buffer while the real buffer level resides unchanged.
4. 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. The smoothened buffer level $\bar{b}_{tk}$ always has a latency (see figure 17) 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.

1. Selecting $b_k^A = 0$ yields no gracefulness at all. Asynchrony immediately affects presentation quality and is soon discovered by a viewer.
2. b_k^A can be dimensioned such that a least the period between the rise of asynchrony and the discovery by the control function is covered.
3. 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.

It is evident that 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 $S(q_{tk})$ is a trade-off between stability and reactivity. The choice or the tuning 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 (35) 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.

Figure 18 shows the results of a simulation[15] for α values between 0.1 and 0.9. The kernel buffer space b_k is varied between 1 and 25 frames, depicted on the x-axis. The plots show the additionally required buffer space b_k^A for covering case 2. For α tending to 0.9, b_k^A increases strongly as the reaction becomes very indolent. Remember that b_k^A is needed twice, above and below the kernel buffer. Thus, e.g. for a kernel buffer of 10 and using $\alpha = 0.9$, 36 extra buffer slots are required on the whole. This corresponds to 360% of the kernel buffer space.

Therefore, an upper bound for α is given by the provided 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 reasonable buffering.

To show the impact of low values of α, we simulated the buffer fill level over a period of 135,000 frames, corresponding to a video with a duration of 90 minutes. In dependency of α, we counted the resynchronization actions due to jitter effects. The computation was made with a buffer space of 25 frames with quarter fill level, initially. We applied a gaussian distribution for the jitter with a confidence of 95% for frames being within their jitter bounds. The assumption of a gaussian distribution has been taken merely to show the effect of different values of α and does not claim to be the real distribution of the jitter. In fact, this distribution is general unknown and highly dependent on the system architecture. Figure 19 shows the simulation results. A number of 1350 resynchroniza-

[15] The simulations throughout this section have been carried out with the Matlab tool [Mat92].

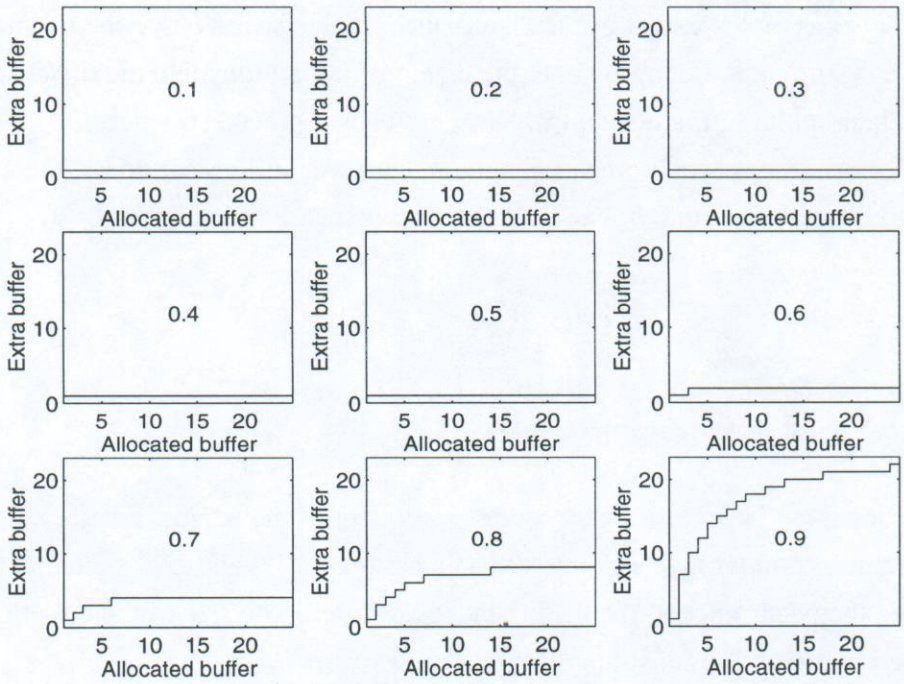

Figure 18: Additional Buffering b_k^A for different values of α.

tion actions over 135,000 frames, for instance, equals 1%.

Values of α between 0.1 and 0.5 lead to a relative high number of unrequired resynchronization actions. From 0.6 up to 0.9, values drop drastically while for 0.8 and 0.9 they become zero. Comparing these results with the buffer requirements computed above, a value of 0.6 or 0.7 for α is a good compromise with respect to the buffer requirements and the number of unnecessary resynchronization actions for the described example.

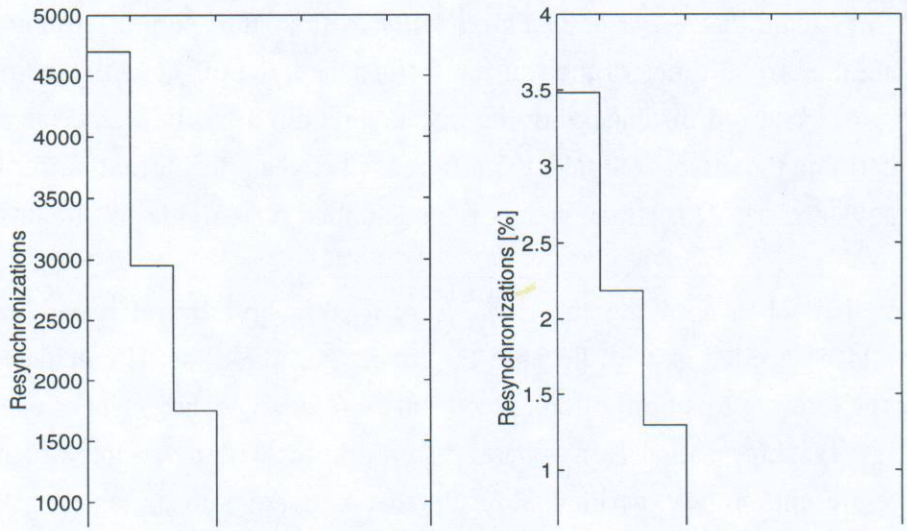

Figure 19: Resynchronization Actions Due to Jitter Effects.

In order to get full gracefulness (case three) we need to cover case two and, moreover,

additional buffering is needed to cover the period until the asynchrony is removed. This strongly depends on the extent of the disturbance. In the easiest case, one resynchronization phase is sufficient. Let o_{tk} be both the extent of asynchrony and the offset sent to the server. When applying the concept of skipping/pausing the server needs $o_{tk} \cdot r/n$ seconds to perform the resynchronization action. Thus, for full gracefulness, the additional buffering beyond case 2 and b_k can be stated in worst case as follows.

$$2 \cdot \left\lceil \left(\bar{d}_k^{new} + \Delta_k + o_{tk} \cdot \frac{n}{r} \right) \cdot \frac{r}{n} \right\rceil \tag{36}$$

(c) Degree o_{tk} of Resynchronization

Resynchronization is performed by sending an offset to the servers where the goal is to move the buffer pointer $\bar{b}_{tk}$ back into the area between UW_k and LW_k such that the situation before the occurrence of the disturbance is restored. 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 by undergoing 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 frames. High values could lead to oscillation because the extent of asynchrony is always overestimated.

When applying the *variable offset* strategy, o_{tk} is varied depending on the extent of the occurred asynchrony. A benefit of the video server array architecture is that we have several substreams building one continuous media stream. This enables us to compare the state of a substream that is *out-of-synchronization* with a substream *in-sync*hronization. Considering the arrival times of the frames, for instance, the offset can be computed by comparing the occurred distance with the distance required by the frame rate. Another way to determine the offset is to take the difference between the current buffer level and the related watermark. This strategy can even be applied for a single substream.

Remember that when applying the *variable offset* strategy several *resynchronization phases* could be needed as well because the time when the offset is calculated (determined by the filtering function) often reflects only a fraction of the extent of asynchrony. Nonetheless, synchronization is in general restored faster with a variable offset. In section 4.5 we presents some experimental results that compare both strategies.

(d) 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 to the watermarks. Again, resynchronization actions may be taken.

R must be chosen sufficiently large that the server can perform the resynchronization, this is, the action must already have taken effect on the client. Selecting R too low leads to numerous unnecessary resynchronization phases where during each phase the extent of asynchrony is overestimated. Take for instance a buffer overflow. Appropriate resynchronization actions are injected which can result in a buffer starvation because of overestimating the asynchrony. Again, resynchronization is started. Thus, low values of R can result in oscillation.

For large values of R values several resynchronization phases are needed as well but the total time of resynchronization becomes unacceptable long. So, in both cases presentation quality might be strongly influenced.

The reaction time on resynchronization measures depends on the extent of asynchrony and the transmission time. Derived from (36), a lower bound for R is given by (37).

$$R \geq \bar{d}_k^{new} + \Delta_k + o_{tk} \cdot \frac{n}{r} \tag{37}$$

4.4.6 Intra-frame Striping

The distributed architecture of the Video Server Array offers two possibilities of distributing or striping frames onto the servers: *inter-frame striping* and *intra-frame striping*. The proposed synchronization scheme is developed under the assumption of inter-frame striping. With little modifications it is equally applicable for intra-frame striping. Employing this striping technique, each frame is divided into *n subframes*. Each of these subframes is stored on one of the n server nodes. The server nodes transfer the subframes such that each piece is received at the client at the same time. The client recombines the pieces to a complete frame and displays it. In contrast to inter-frame striping the temporal relationship between the striping blocks is *equality*, i.e. each piece is expected at the same time on the sink site. In contrast to inter-frame striping, each server schedules the frames with the same rate as the client. [Ber95b]

Suppose all introduced variables refer to subframes. To start playout in a synchronized manner, the shifting of r^{-1} between the substreams imposed by the frame rate must be dropped. So, equations (9), (10) and (11) given by the start-up protocol have to be modified as follows.

$$t_0 = max\{t_{ref} + d_i | i \in I_0\} \tag{38}$$

$$\upsilon = \{j \in I | t_{ref} + d_j = t_0\} \tag{39}$$

$$s_i^c = s_i + d^{max} + \delta_{\upsilon i} \qquad \forall i \in I_o \tag{40}$$

The derived buffer requirements to attain intra- and inter-stream synchronization in model 2 can be applied without any modification, except for the frame rate. For the play-out deadline, theorem 3 is to be applied instead of lemma 1. The scheme of buffer level control remains unaffected by the used striping technique.

4.4.7 Synchronization of Multiple Streams

The problem of synchronizing multiple streams occurs if several related synchronization groups are to be synchronized, e.g. an audio group and a video group. Each group consists of an arbitrary number of substreams. We assume the groups to be presented simultaneously, i.e. frame 0 of one group refers to frame 0 of another group. To solve this synchronization problem with the proposed scheme we suggest to sum up all synchronization groups in a *super group*. The calculations of model 2 can be applied to this super group without any modifications.

The start-up protocol has to be modified little. It must be take into account that there exists a starting points t_0 for each synchronization group. Taking the maximum of these t_0 values as earliest starting time for all groups allows to initiate start-up of the servers in a synchronized fashion. The problem is complicated if the substreams of each group are sampled at a different rate. In this case, we can get started once in a synchronized manner but further synchronizations due to VCR functions can not be performed with the generalized start-up protocol because sequence numbers are not related. We propose two techniques to cope with this problem. First, the *greatest common divisor* (GCD) of the frame rates of all synchronization groups can be taken as a basis for determining the sequence numbers [Ran93]. This mostly implies a modification of the sequence numbers after having captured a media stream. Second, a *mapping of sequence numbers* can be employed. The synchronization group with the highest frame rate serves as a reference. For instance, frame number 6 of a group with a rate of 12 fps is mapped to frame number 12 of a group with a rate of 24 fps. Depending on the frame rates this technique introduces inaccuracies by rounding. The detailed examination of the synchronization of multiple streams within the Video Server Array is subject of future work.

4.5 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 chapter 5.

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

- A "Bitburger" commercial, sampled at a rate of 16 fps with a total length of 462 frames.
- A scene from the production "Seaquest", sampled at a rate of 16 fps with a total length of 6710 frames.

For the first experiment, we measured the inter-arrival times of frames for the video "Bitburger" and plotted the cumulative values along with the consumption rate as shown in figure 20.

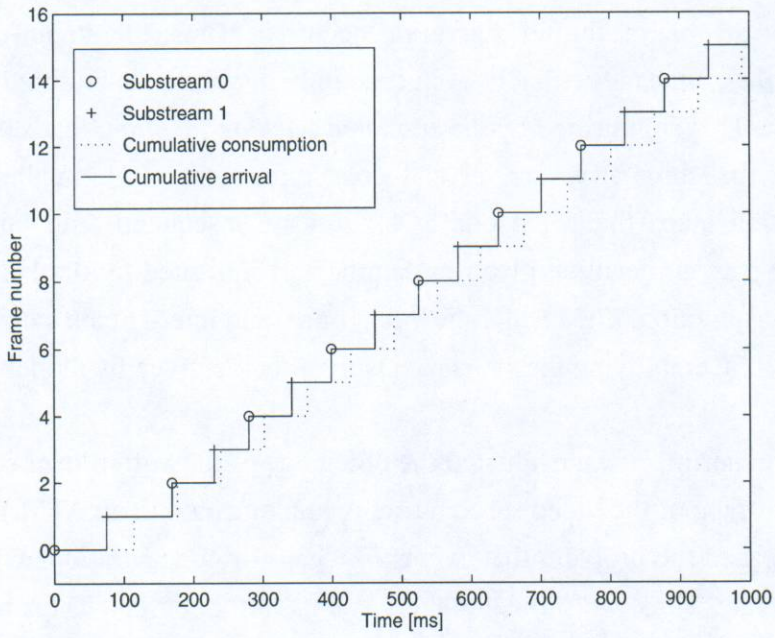

Figure 20: Cumulative Arrival and Consumption for two Substreams.

The experiment was conducted with two substreams. For substream 0 and substream 1 we measured jitter values of 26 ms and 24 ms, respectively (see table 3). According to theorem 4, 5 and 6, 2 buffer slots are allocated for each substream, including a buffer slot

Number of substream	jitter [ms]	buffers [frames]	start-up latency[a] [ms]	roundtrip delay [ms]
0	26	2	252	33
1	24	2	314.5	52

Table 3: Experimental Results for Intra- and Inter-Stream Synchronization.

a The start-up latency is usually defined as the delay between user interaction and visible feedback [Den95].

to read a frame from the network. The shifting of two milliseconds between the two substreams is almost neglectable. The start-up protocol leads to a start-up latency of 252 ms for the first substream and 314.5 ms for the second substream. These times include an additional, overall charge of 200 ms for processing time. Thus, we can see clearly that the maximum roundtrip delay of 52 ms determines the starting time of the first server. The second server starts exactly 62.5 ms later, according to the frame rate of 16 fps. These results prove that the start-up protocol is performed with a high accuracy.

Consider now figure 20, showing the cumulative arrival times of the first 1000 ms of the "Bitburger" commercial. The x-axis displays the elapsed time while the y-axis shows the frame number. The cumulative arrivals of frames of both substreams never cross the cumulative consumption, i.e. the cumulative arrival stays always above the consumption, indicating that at no time buffer starvation occurred. Thus, the stream is played out smoothly. Further, we can see that at each time only one frame is buffered for each substream at most. This is indicated by the so-called *backlog function*, that states the difference between the cumulative arrival and consumption [Kni94]. In the example the backlog function takes the value one at all time. Consequently, no buffer overflow occurred. The playout deadline given by lemma 1 is indicated by the beginning of the cumulative consumption. The results show that intra- and inter-stream synchronization is performed well by employing the synchronization scheme given by models 1 and 2.

In the second experiment, 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 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. We conducted this experiment again with the "Bitburger" commercial striped only onto a single server. The following parameters have been used:

- Smoothing factor for the geometric weighting function: $\alpha = 0.7$
- Amount of injected asynchrony[16]: -8, -4, +4, +8
- 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 constantly to 1. According to the previous experiment we allocated two buffer slots for the substream. This corresponds to the kernel buffer b_k defined in section 4.4.5.2. Further, for the additional buffering b_k^A, we were using three buffer slots twice, above and below b_k. Consider now figure 21, showing the virtual buffer level and the filtered buffer level over time for the resynchronization of a concentration of eight frames. The x-axis shows the virtual buffer level while the y-axis denotes the consumption period. The upper bound 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 so to calculate the virtual buffer. Figure 21 shows the course of resynchronization if the fixed offset strategy is employed.

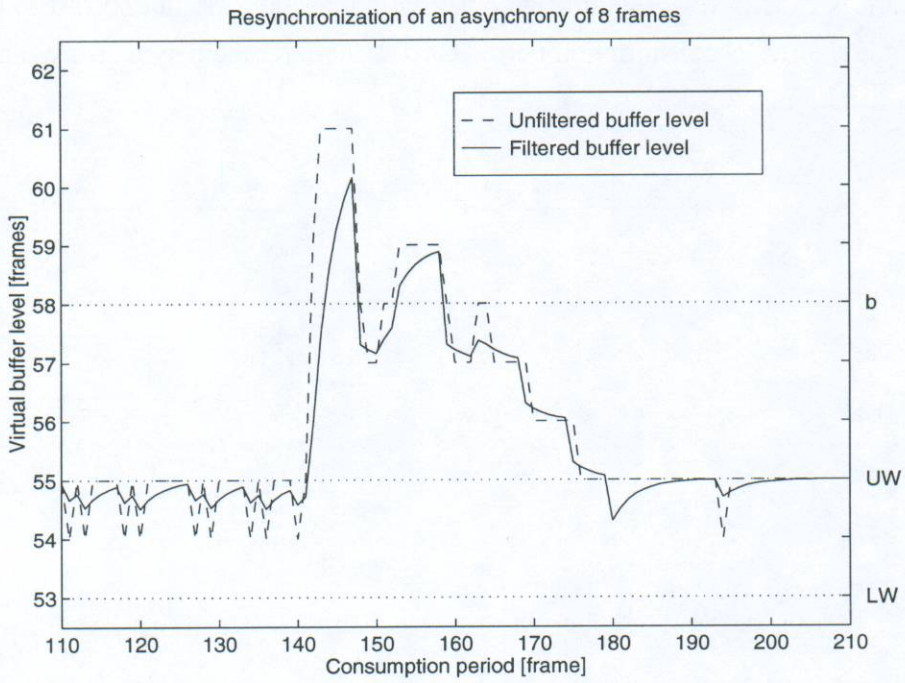

Figure 21: Resynchronization with the Fixed Offset Strategy.

The first resynchronization phase is entered exactly during consumption period 142 when the filtered buffer level crosses the upper watermark. The virtual buffer level rises

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

up to 61, that is, four frames are discarded. The lost of four frames could be perceived during playback. The client then sends an offset of -1 to the server. After the end of the first resynchronization phase the filtered buffer level is reset to the virtual buffer level as indicated by the abrupt decrease of the filtered buffer level depicted in figure 21. 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 can therefore conclude a total duration of 38 consumption periods for the resynchronization of an asynchrony of eight frames with the fixed offset strategy. This corresponds to 2375 ms.

In contrast, we now consider the same situation with the variable offset strategy. The course of the filtered and unfiltered buffer level is depicted in figure 22. 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. The buffer level falls already below *UW* for a short period of time. Now, two additional resynchronization phases are undergone until synchrony is restored. In each phase an offset of -2 is sent to the server. Again, phases can be observed by the peaks of the filtered buffer level shown in 22. Synchronization is exactly restored during consumption period 149. In contrast to the fixed offset strategy, only 19 consumption periods are needed to regain synchrony. This corresponds to 1187.5 ms.

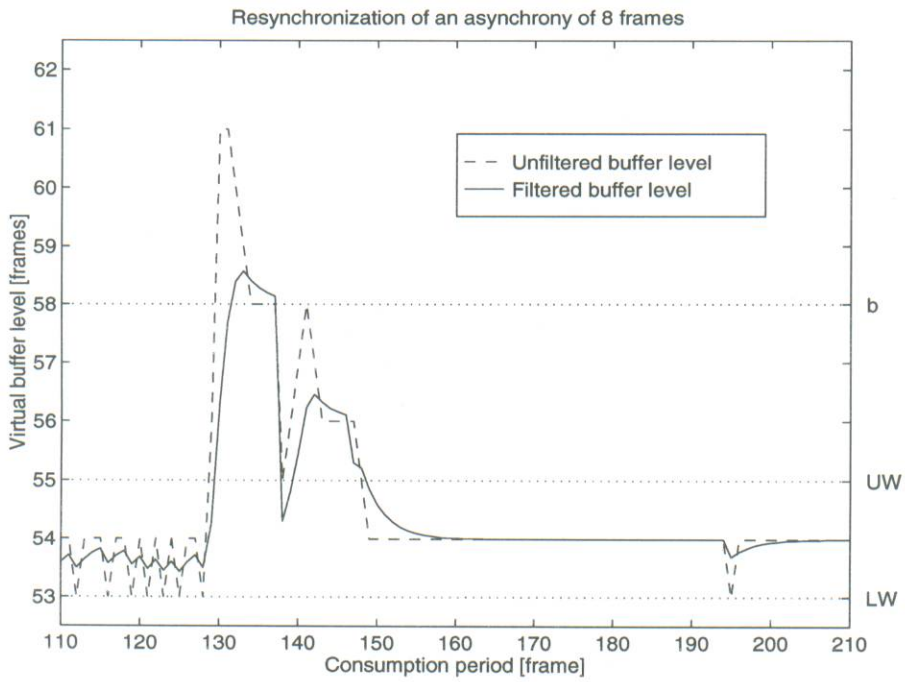

Figure 22: Resynchronization with the Variable Offset Strategy.

To obtain more representative values about the total duration of resynchronization in dependency of the applied strategy, we performed a third experiment. This time, we took

a video of longer duration. During the playback of "Seaquest" 50 resynchronizations are undergone. We estimated the duration by taking the mean value of the duration sum. Table 4 presents the experimental results for different sizes of asynchrony.

Asynchrony	Fixed offset		Variable offset		[%][a]
	Mean [ms]	Variance [ms]	Mean [ms]	Variance [ms]	
-4	1,143.80	223.13	773.75	78.95	67.6
4	1,327.50	250.50	707.50	139.72	53.3
-8	1,223.80	310.60	665.00	123.56	54.3
8	2515.00	789.21	1,081.20	202.95	43.0

Table 4: Mean and Variance of the Resynchronization Duration.

a The percentage values compare the two strategies and are calculated with respect to the fixed offset strategy results.

If applying the variable offset strategy, the duration of synchronization can be more than halved compared to the fixed offset strategy, indicated by the 43% for an asynchrony of 8 frames. The results also show clearly that resynchronization with a variable offset becomes even more efficient for larger asynchronies because adoption is performed faster. For negative asynchronies or gaps, respectively, we can see that the gain with the variable offset is not as high as for positive asynchronies. This can be explained by how the asynchony is produced. While concentrations are introduced by sending a specified amount of data at once, gaps are produced by just skipping a number of frames at the server. So, a concentration arises immediately and can therefore be removed faster. Gaps arise slowly because the buffer at the client has is emptied only every presentation cycle. Thus, resynchronization for gaps is done slower when comparing the two strategies.

The conducted experiments prove the efficiency of the buffer level control mechanism. Further, they indicate that a variable offset strategy restores synchrony much more faster than a fixed offset strategy. Further plots for different sizes of asynchrony can be found in appendix C.

5 Design of a Video Server Array Prototype

5.1 General System Architecture

We have implemented a prototypical video server based on the Video Server Array as described in chapter 2 [Den94]. Our system configuration consists of a set of server nodes, to which a number of clients is connected via an ATM switch. Each time a client is requesting the playback of a video, point-to-point connections are established between the involved server nodes and the client. To facilitate the request mechanism we additionally introduced a meta server in the architecture. The meta server has complete knowledge of the location of the video material and it provides a directory service to the clients. Figure 23 demonstrates the system architecture.

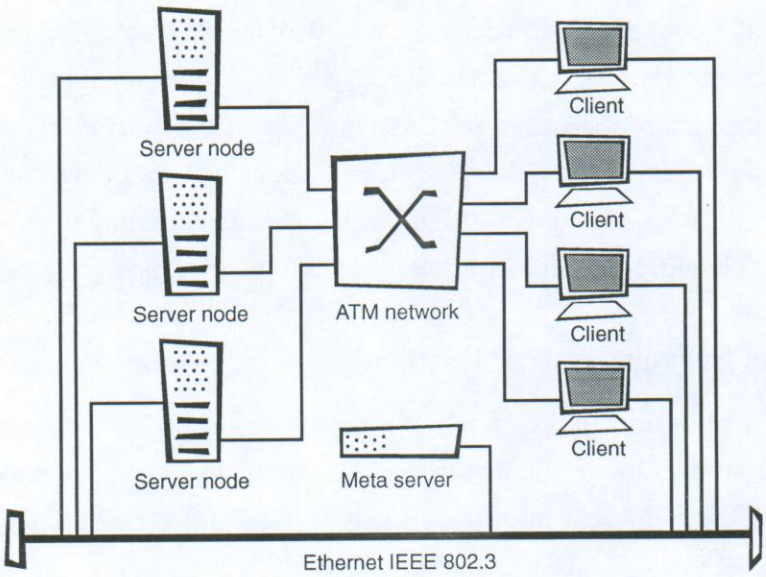

Figure 23: Configuration of the Prototypical Video Server Array.

The system was developed on SUN Sparc 2 and Sparc 10 workstations for the server nodes, the meta server, and the client. All stations are connected via a 10 Mbit Ethernet, server and client nodes are additionally connected via an ASX-200 Fore Systems ATM switch. Stations are equipped either with Fore Systems SBA-200 or SBA-100 ATM

SBus adapters. Client nodes are supplied with Parallax XVideo boards that provide real-time motion JPEG codec[17] [Lai94]. As secondary storage, each server nodes is equipped with a dedicated Micropolis 4110 AV hard disks that is accessed directly by a raw disk interface so to avoid the UNIX file system's overhead.

The communication with the meta server is based on the TCP/IP protocol over low bandwidth Ethernet because only low amounts of non time-critical data are transferred. The continuous media data that is consumed at very high rates is transferred via the ATM protocol. The ATM network is accessed by a UNIX UDP socket interface [Ste90b] that is mapped to the ATM adaptation layers (AAL). Clients are provided with a user interface that has been implemented with the Tcl/Tk toolkit [Ous94] whereas all other components of the prototype have been implemented in the C++ programming language [Sch94], [Ker90].

The following sections describe design issues and the implementation of each component of the prototype system. To begin with, we explain the protocol flow between all involved components during the establishment of a playback session.

5.2 Application Protocol

We distinguish between two different protocols. First, the *meta server control protocol* that is handled via connection-oriented TCP/IP over Ethernet. The protocol offers a directory service to the client nodes and realizes session establishment between client and server nodes, acting as a negotiator. Second, the *video transmission protocol* based on connectionless UDP over ATM provides the transmission of the continuous media data as well as the exchange of control messages, e.g. for initiating playout in a synchronized manner. The different protocol flows are outlined in figure 23.

5.2.1 Meta Server Control Protocol

Whenever a client desires the playback of a video, it contacts the meta server by issuing a META_INFO_REQ[18] so to obtain information about the available videos. The meta server delivers the requested information, e.g. title, abstract, or duration, by sending a META_INFO_RSP. Once the user has selected a video, the client requests the playback by sending a META_PLAY_REQ to the meta server. The meta server either refuses the request by sending a negative META_PLAY_RSP back the client, e.g. if no servers are connected

[17] The quality provided by the XVideo adapter corresponds to true color "Quarter PAL" which is comparable to VHS quality. Frame sizes typically vary between 10 and 14 Kbytes.

[18] The names of the service primitives correspond to the names used in the implementation.

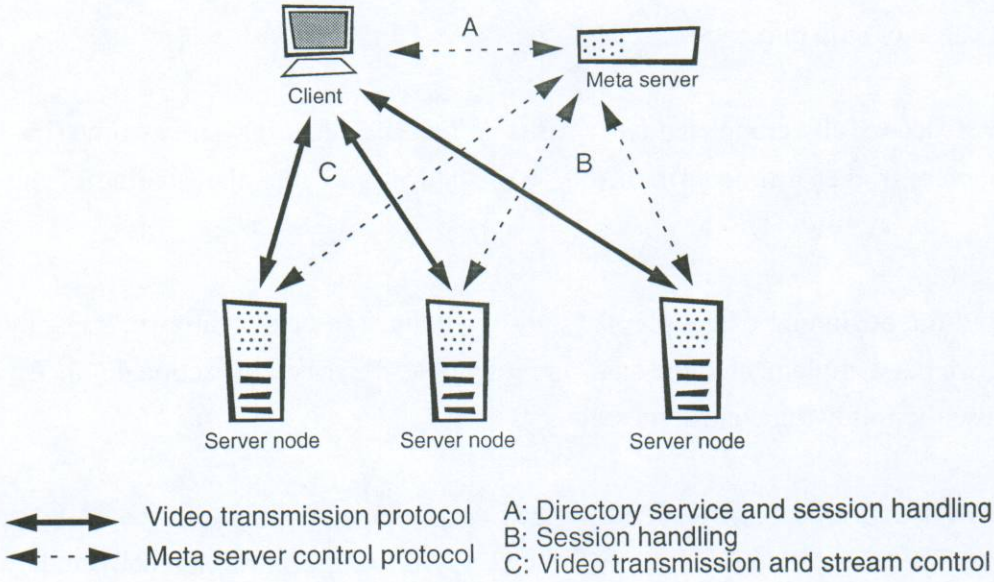

Figure 24: Protocol Model in the Prototype Video Server Array.

to the metaserver, or invokes a complex *two phase commit* protocol. This protocol is needed since we can not guarantee a priori that all server nodes accept the request. Initially, the meta server notifies the involved server nodes as well as the client by sending a TWO_P_PREPARE. Each server and the client then allocate the resources for the playback and servers additionally prefetche data from the disk. If successful, the nodes reply with a TWO_P_READY. Otherwise, they issue a TWO_ P_ABORT. These messages also carry information about the location of the servers and the port numbers.

The meta server collects all replies, and on success, the second phase is started by sending a TWO_P_COMMIT to all involved nodes so to notify server nodes and client of the location and port number of each other. Otherwise the protocol is terminated by issuing a TWO_P_ABORT.

When receiving the commit message, the client has complete knowledge of all video servers that will deliver the video and vice versa. The commit message is acknowledged by sending a TWO_P_READY back to the meta server. All involved nodes are now ready for playback.

5.2.2 Video Transmission Protocol

The video transmission protocol is employed for both the transmission of frames and the exchange of control messages concerning the playback of the video stream.[19] Control

[19] This can be thought of as a kind of *inband signaling* because we use the same network I/O device for the transmission of frames as well as for controlling the server nodes.

messages are used, for instance, to get the server nodes started in a synchronized manner, or to pause or stop playback.

After a successfully completed two phase commit, the client may at any time (i.e. before the servers trigger a timeout) tell the server nodes[20], to start the sending of the video data.

Prior to the beginning of the playout the servers have to be synchronized. For this purpose, we have implemented the start-up protocol as described in section 4.4.3. We therefore use the following control messages[21]:

- CONTROL_ADJUST corresponds to the *Eval_Request* message of the start-up protocol. The transmitted PDU requests the first frame of the substream delivered by each server. Its purpose is to calculate the roundtrip delay on each server connection.
- CONTROL_START corresponds to the *Sync_Request* message of the start-up protocol. After having received all the first frames from each server, the client calculates the future starting times of the servers and propagates them back by sending a CONTROL_START message.

Our implementation further allows for the VCR functions pause, stop, fast forward, and replay which are implemented by the following control messages[22]:

- CONTROL_STOP issued by the client terminates the playback of the current video. The video is removed out of the service queue in the server nodes.
- CONTROL_PAUSE causes the server nodes to immediately pause the scheduling of the substream.
- CONTROL_CONTINUE: After having paused, the playout of the servers has to synchronized again by employing the start-up protocol (see section 4.4.3.3). The CONTROL_CONTINUE corresponds to the CONTROL_START message.
- CONTROL_FFORWARD: This control messages also issued by the client causes the server nodes to skip multiples of frames during scheduling in order to achieve the VCR function fast forward.

The calculation of elastic buffers on the client site is based on the assumed jitter. We provided our architecture with a possibility to experimentally measure the overall delay jit-

---

[20] This process is initiated by pressing the *play* button in the user interface.

[21] For the detailed protocol flow refer to section 4.4.3.2.

[22] Notice that each control message is sent to all involved server nodes.

ter as defined in section 4.4.4 prior to the beginning of the start-up protocol. For this purpose we use the following control message:

- CONTROL_TRIGGER issued by the client to all involved server nodes starts a so-called trigger phase. On receiving the message, server nodes start the scheduling of substreams with the maximum frame rate until they receive a control stop message.[23]

We also implemented the buffer level control described in section 4.4.5. A substream that eventually falls out-of-synchronization during the playback of the video can be re-synchronized by sending a

- CONTROL_OFFSET message to the respective server node. The related PDU carries either a positive or negative offset value enforcing the server to skip media data or to pause for a period of time, respectively.

We further use the following two messages:

- CONTROL_ABORT equals a CONTROL_STOP except that it is issued in case of an error.
- CONTROL_END is issued by the server nodes, indicating the end of the current substream.

Media data is transmitted straightforwardly in fixed size UDP datagrams. The data structure of the transmitted PDU for both control messages and frames is described in appendix E. Furthermore, an example for a complete protocol flow in presented in appendix D.

5.3 Server

5.3.1 Design Parameters

Our video server consists of an array of server nodes as described in section 2.2. Frames are striped across all server nodes and each server contributes to the playback with its share of video information. Notice that the number of servers *n,* across which a video is distributed, is not restricted in our approach. Several design parameters have to be decided.

1. The *disk block size* determines the retrieval units from secondary storage. The block size must be chosen by trading off buffer requirements against available disk

[23] Note that in a trigger phase the server do not advance in reading video data from disk. They schedule repeatedly the current contents of the prefetched data out of the buffer.

bandwidth or achievable throughput, respectively. The retrieval time of a disk block comprises a variable seek time, the rotational latency, and the time to transfer the block [Ber95a]. Since the variable seek time and the rotational latency are independent of the amount of transferred data, we have to chose a block size such that this overhead is minimized.

2. The *disk scheduling algorithm* determines how and in which order data for concurrent read request are read from a hard disk. An algorithm has to be chosen such that the real-time requirements introduced by video and audio streams can be satisfied.

3. The *striping block size* defines the number of contiguous frames that is stored in its entirety on a single server node. In section 2.2, we already mentioned the two distinctive strategies. First, it is conceivable to store *entire frames* on server nodes in a round-robin fashion. This method is called *single-frame striping* or *inter-frame striping*, respectively. Second, only parts of each frame, called *subframes*, can be stored on server nodes, what is called *subframe striping* or *intra-frame* striping. The method of distributing a video across the server nodes determines the temporal relationship that is required for the synchronization (see sections 4.3 and 4.4.6).

4. The *network transfer mode* defines whether the video server sends the video data in bursts or continuously in time. In *burst mode*, the data are sent in periodic bursts, with the burst transmission rate being higher than the average consumption rate. On the other hand, in *continuous mode*, transmission of media data is uniformly distributed over an interval determined by the playback rate.[Ber95b]

In the prototype implementation of the Video Server Array we chose a *disk block size* of 450 kilobyte because for the Micropolis AV 4110, this block size corresponds to the data contained in one disk cylinder. Thus, multiple track-to-track seeks are avoided during read operation. The *disk scheduling* is based on the first come first served scheme that is actually less suited for meeting real-time constraints. Disk scheduling has to be improved by future work. For the *striping technique*, we chose inter-frame striping, that is, entire frames are distributed across the server nodes in a round robin manner as described in 4.4.4.1. The synchronization scheme proposed in chapter 4 is suitable for inter-frame striping and intra-frame striping as well. Finally, we employ the *burst network transfer mode* where a network block size corresponds to one frame.

5.3.2 Implementation

A video server node has to satisfy two main tasks. First, the retrieval of media data from the storage devices. Second, the transmission of the retrieved data to the client. While the scheduling of the data transfer to the network is determined by the frame rate of the requested video material, the retrieval of media data from a storage device is usually

scheduled much less frequently. Both tasks are subject to stringent real-time constraints and must be therefore logically uncoupled.

Consequently, each one of the two tasks has been implemented by a separate process, as depicted in figure 25. The first process, called *stream manager,* transmits the video data to the client and handles the control communication with the meta server and the client. The network communication facility is provided by a stream manager object. The stream manger process also includes a real-time, rate-monotonic scheduler so to provide timely delivery of the continuous media data. The *disk manager* which is the second process, is forked out from the stream manager process and operates under its parent control. The *disk manager* is responsible for the retrieval of media data from the hard disk.

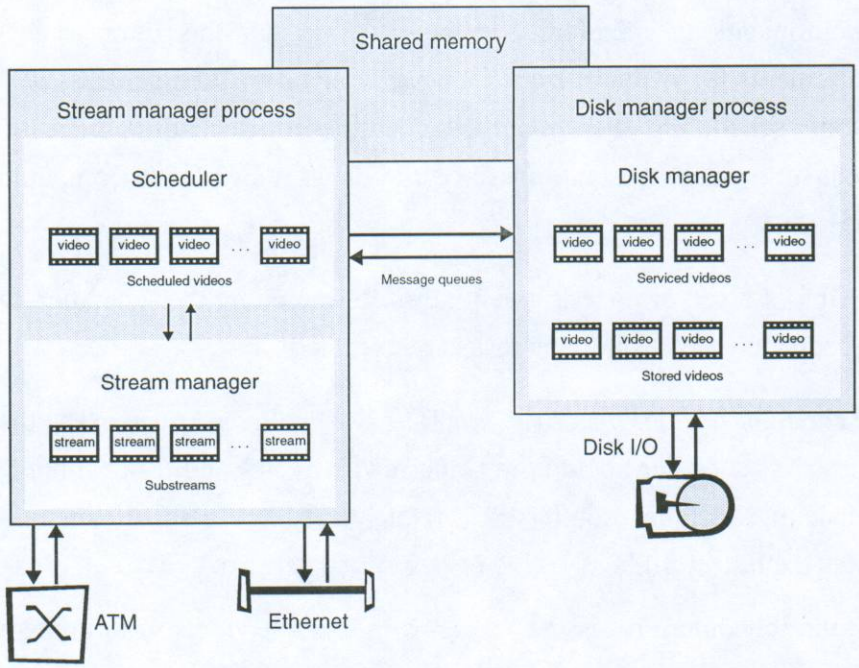

Figure 25: Architecture of the Video Server Node.

The communication between the two processes is realized by a *shared memory* pool for the media data, and a message queue for the commands. Shared memory of the size of two disk blocks is allocated for each serviced substream. The two shared memory buffers are used in a cyclic manner, i.e the data from one buffer, which was previously filled by the disk manager, is read out and transferred to the client while the other buffer is refilled by the disk manager.

5.3.2.1 Real-time Scheduler

Each video server node contains a real-time scheduler that assures the continuous play-out of the video. The scheduler operates in fixed scheduling cycles of 40 ms. Despite of the fixed cycle duration, it can still support arbitrary frame rates up to 25 fps[24]. During each cycle, the scheduler computes for all serviced substreams the amount of media data that has to be sent to the respective client, i.e. it computes whether or not the next video frame becomes due during the current cycle. If the deadline of the frame is met during the current cycle, the scheduler notifies the stream manager to deliver the frame to the client. Hence, the scheduler introduces gaps of multiples of 40 ms for substreams with a frame rate below 25 fps, so to wait until the next frame again falls into a scheduling cycle. Obviously, this technique injects some end-system jitter that must be smoothed out by buffering at the client. Jitter affects intra-stream synchronization as well as inter-stream synchronization. Therefore, we have to consider this jitter when calculating buffer requirements for synchronization (see section 4.4.4). Nonetheless, we favored the rate-monotonic scheduling over an earliest-deadline-first scheduling because EDF does not offer real-time guarantees and possibly introduces a significant computational overhead [Ste95].

The processing of a server node is essentially determined by the scheduler. During each cycle of the scheduler, the following tasks are executed:

1. The scheduler handles *messages from the disk manager process.* The disk manager notifies the scheduler about completed read requests and disk failures. Whenever the disk manager has completed a read request for a substream, the scheduler updates the buffer fill level of the respective substream.
2. Next, the scheduler processes *messages from video clients* so to immediately react to client interaction, such as play, pause, etc.
3. The scheduler now inspects all serviced videos in a round-robin manner. If the frame of a substream is due during the current cycle the respective *video data is sent* to the client. Each frame is provided with a time stamp needed for the start-up protocol at the client site and for the calculation of jitter. If a buffer segment runs empty, the scheduler immediately passes a new read request to the disk manager in order to stimulate the refill of the respective segment.
4. Finally, *messages from the meta server* are handled to allow for the admission of new clients.

---

[24] Notice that if videos are distributed across several nodes in a server array, a substream frame rate of 25 fps will be rather unlikely since substreams need only be played back at a fraction of the video's original frame rate.

5.3.2.2 Stream Manager

Each server node is equipped with a stream manager object that handles all network I/O. The object includes a TCP socket for the connection to the meta server, and a UDP socket for each client connection. Further, it offers methods to send frames to video clients and to exchange control messages with clients and the meta server. Since the size of UDP datagrams is restricted, the stream manager further packetizes video frames that are to be sent to the clients.

5.3.2.3 Disk Manager

The disk manager is located in another process that also attaches to the shared memory pool allocated by the stream manager process. The disk manager accesses the dedicated 4110 AV hard disk via a raw I/O device so to retrieve media data that is requested by the scheduler. The media data is retrieved into the shared memory buffer space for the *serviced videos* using FCFS. Additionally, the disk manager stores all information regarding the videos' storage location on the device, as indicated by the list of *stored videos* in figure 25.

5.4 Client

5.4.1 Architectural Requirements

The video client is the element in our architecture that allows for accessing the video on-demand service offered by the server array. It has to coordinate the playback of a video stream. The design parameters that have been chosen for the Video Server Array also affect the design of the client. In particular, we can identify the following main tasks that must be accomplished by a client:

1. The video client has to provide a *graphical user interface* (GUI) that enables the viewer to select a video, to retrieve further information, and to execute VCR functions such as play, pause, stop, and fast forward.
2. The *communication* between meta server, server nodes, and the user must be managed. Specifically, the client has to provide a mechanism for the depacketization of UDP packets back to complete frames.
3. The *timely presentation* of the video data must be guaranteed by a real-time scheduling mechanism.
4. Tightly coupled with the presentation of a video is the *synchronization* of the different substreams that constitute the entire media stream. Both intra-stream synchronization for each substream and inter-stream synchronization between the

substreams must be carried out by the client during the playback of the video. Furthermore, the client must provide mechanisms to start-up the server nodes in a synchronized manner and to allow for resynchronization during playback if a substream gets out of synchronization.

5.4.2 Implementation

The user interaction with the GUI is an activity that must be provided independent of the currently displayed video stream. We consequently split up the video client in two separate processes to allow for full asynchronous interaction. The video client consists of a GUI process and a client process as demonstrated in figure 25. The client process is completely event-driven and communicates with the GUI via message queues. Further, it is designed to play several videos simultaneously which are administrated in the *serviced videos* list.

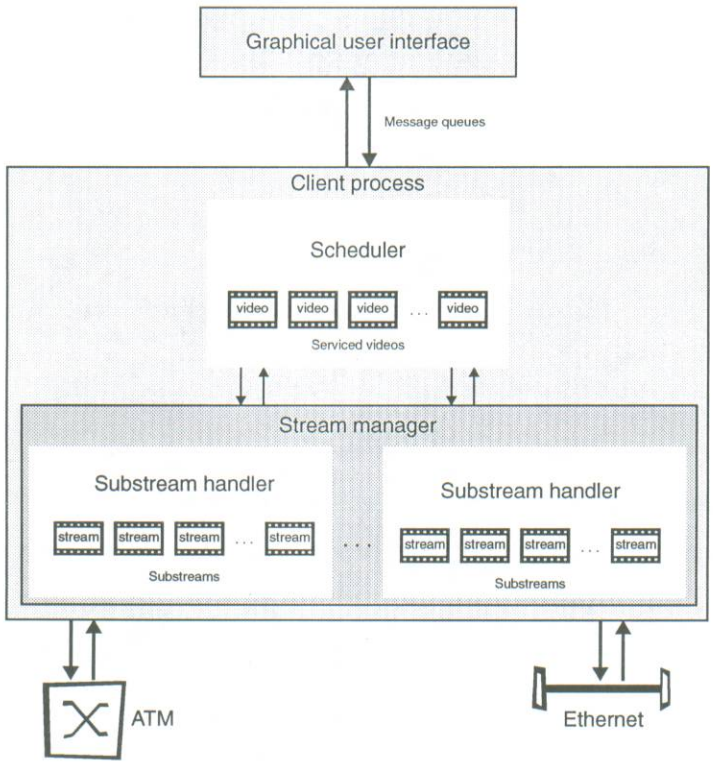

Figure 26: Architecture of the Video Client.

5.4.2.1 Client Process

The client process consists of a *stream manager* that handles all the network communication, i.e. control messages from the video server and the meta server, and video data from the server nodes. The client also includes a real-time *scheduler* in order to ensure the timely presentation of the streams. The processing in the client is determined by a

main dispatch loop that provides the handling of events from the server nodes, the meta server, and the GUI and by a scheduling function that is controlled by the system interval timer. During each dispatch loop the client process carries out several activities:

1. The client browses through the list of serviced videos and checks the *server node connections* for incoming traffic. Arriving frames are depacketized and inserted into the respective buffer queue. Control messages from server nodes and X events are handled. Furthermore, the state of each video is checked and eventually state transitions are performed.
2. User interface input is checked and handled.
3. Input from the meta server is checked and handled.

(a) Stream Manager

The client is designed to support multiple video streams simultaneously. For this purpose, the stream manager creates a substream handler for each video stream. The substream handler encloses the UDP sockets required for the substreams. Furthermore, each substream handler provides the facilities and mechanisms for the synchronization of the substreams.

Each substream handler allocates buffer space according to the jitter bounds that have been specified either by the user or by undergoing a jitter phase prior to the beginning of playout. Corresponding to model 2 (see section 4.4.4), a separate buffer queue is allocated for each substream. The substream buffer queues are enclosed within a class called *StreamBuffer* that masks the existence of several buffer queues, i.e. whenever the scheduler retrieves a frame from the buffer, the StreamBuffer performs a mapping to the respective substream buffer by delivering the next frame. For each substream buffer, we also implemented the concept of the virtual buffer as proposed in section 4.4.5, this is, we defined a virtual buffer range as well as lower and upper watermarks. Additionally, the substream handler offers functions to perform the start-up calculation and the resynchronization during playback. For the detailed structure of the buffer queues, refer to appendix F.

During each dispatch loop the client checks whether or not the playout deadline given by lemma 1 in section 4.4.4 is reached. For this purpose a respective function has been implemented in the substream handler. Once the playout deadline is reached, smooth playout can be ensured, i.e. the consumption of frames from the buffer can be started by the scheduler.

(b) Scheduler

The scheduler assures the real-time playout of the video stream. It operates at a constant rate driven by the system interval timer. The rate corresponds to the video's frame rate. Each cycle a certain frame number is expected to be played out. The scheduler inspects the buffer and compares the frame number of the first frame in the buffer to the frame number that is currently due. If numbers are equal, the respective frame is removed from the buffer queue and displayed. Otherwise, we have to distinguish two cases:

1. If the number of the first frame in the buffer queue is lower than the current frame number, the related frame arrived too late and is discarded.[25]
2. If the number of the first frame in the buffer queue is larger than the current frame number, frame losses occurred or the respective frame will arrive too late. To maintain continuity of playback, the last frame is displayed again.[26]

5.4.2.2 Graphical User Interface

The graphical user interface *vplayer* has been developed by using the Tcl/Tk scripting language. The GUI launches the client process and communicates with the client via the standard I/O command pipeline. The communication between GUI and the client is completely asynchronous and triggered by user-defined signals.

The interface enables a user to query video information such as the subject, title, the playback duration, etc., from the meta server. During playback of the video, a viewer can operate the VCR functions play, pause, stop, fast forward, and replay. The look of the user interface is demonstrated in figure 27.

The *Video* menu allows for querying informations from the meta server. A *Select* window is popped up and the user may now chose a video to be played back. Note that the embraced numbers behind the video titles in the select window refer to the number of video server nodes across which the video is distributed. Within the *Audio* menu, the user can select an audio device. The *Options* menu allows to specify values for the jitter bounds on each network connection. Corresponding to these values, buffers are allocated for each substream. Instead of user-provided jitter bounds, a user may also leave it to the client to evaluate jitter values prior to the beginning of playback. Note that the playout of

---

[25] Notice that within one cycle all frame numbers lower than the current number are discarded.

[26] Frame losses may occur sporadically due to transmission errors but also periodically due to a server shutdown, i.e. every *n-th* frame is missing. The more server nodes are involved in the playback of one stream the less a viewer will notice missing frames. This kind of fault tolerance is a benefit of the Video Server Array.

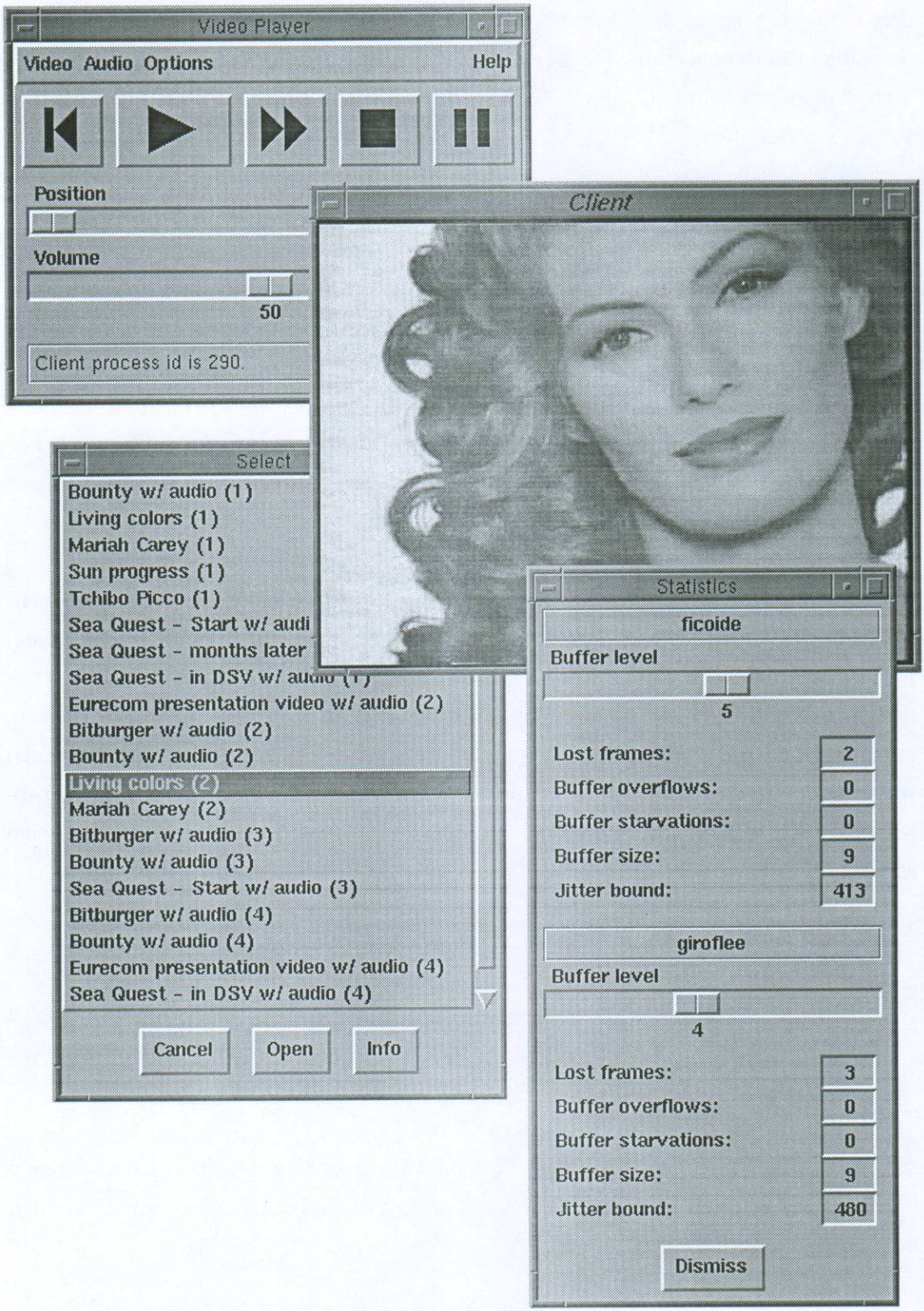

Figure 27: Graphical User Interface.

the video stream is additionally delayed for the duration of the triggering phase. Furthermore, the user has the option to enable or disable the display of specific substreams denoted by the server node's name so to demonstrate the effect of a server shutdown. Statistical values concerning the currently played video stream can be displayed in a separate window as shown in figure 27. A slider shows the current fill level of each substream's buffer queue.[27]

5.5 Metaserver

Most database systems employ a *metadata* mechanism in order to simplify the request mechanism [Row94], [Lit94]. Metadata contain information about data, e.g. location or structure of the data. Employing this scheme within a video server architecture allows for uncoupling the data delivery function of a video server from the database management functions provided by a meta server. So, with respect to the Video Server Array, this means that metadata contain information about the location as well as characteristics of the video data stored in the server array.

In our architecture we adopt to a hierarchical metadata approach where a dedicated, centralized meta server keeps all attributes associated with a complete video Attributes are, for instance, the name, the subject, the frame rate, the resolution, etc., but also information about location, i.e. which nodes of the server array store parts of the video. The data structure of the meta information kept by the meta server is shown in appendix G. Each server nodes in the array only keeps information concerning the management of substreams, such as their location on storage devices associated with the node. The meta server provides two services:

1. A *directory service* allows a client to retrieve information about the stored material.
2. If a client desires the playback of a video, the meta server offers the service of a *session establishment* between client and involved server nodes by coordinating a two phase commit protocol.

Acting as a negotiator during session establishment provides additional transparency during admission and liberates the client from knowing about the location of a video among the nodes of the server array.

The meta server is the leanest process in our architecture. It could be implemented in a

[27] Note that the displayed buffer level refers to the real buffer level.

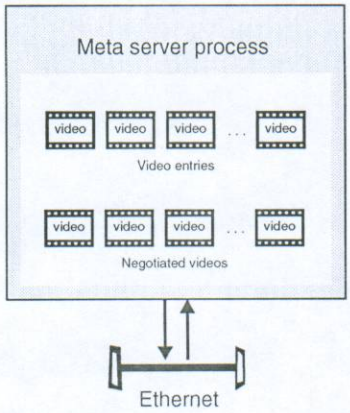

Figure 28: Architecture of the Meta Server.

straight forward manner as indicated figure 28. TCP connections via Ethernet are maintained to all nodes of the Video Server Array as well as to all clients. Meta information is hold in objects called *VideoEntry* derived from *VideoInfo* (see appendix G). Finally, since the main function of the meta server is to negotiate between the Video Server Array and the client for the reason of session establishment, it keeps track of the progress of the two phase commit protocol in a list (*Negotiated videos*) of the currently executed jobs.

Appendix

A Start-Up Protocol Algorithm

(a) Client *D*

$I_0 = \{0,..,n-1\}$
Complete = **FALSE**;
$a_i = 0,\ \forall i \in I_0$;
WHLE $(t < t_{start})$ **DO WAIT**; /* t returns the local time */
SEND *Eval_Request(i)* **TO** $S_i,\ \forall i \in I_0$;
WHILE NOT Complete **DO BEGIN**
    IF RECEIVED *frame(i)* **FROM** S_i **THEN** $a_i = t$;
    IF $(a_i>0,\ \forall i \in I_0)$ **THEN** Complete = **TRUE**;
END;
$t_{ref} = max\{a_i | i \in I_0\}$;
$d_i = a_i - t_{start} \quad \forall i \in I_0$;
$\delta_{ij} = a_i - a_j \quad \forall i,j \in I_0$;
$t_0 = max\{t_{ref} + d_i - i \cdot r^{-1} | i \in I_0\}$;
$\upsilon = \{j \in I_0 | t_{ref} + d_j - j \cdot r^{-1} = t_0\}$;
$s_i^c = s_i + d^{max} + \delta_{\upsilon i} + (i - \nu) \cdot r^{-1}, \qquad \forall i \in I_o$;
SEND *Sync_Request(i,* s_i^c *)* TO $S_i,\ \forall i \in I_0$;
StartReceivingFrames(); /* Frames are received and displayed */

(b) Server S_i

IF RECEIVED *Eval_Request(i)* **FROM** *D* **THEN BEGIN**
    $s_i = t$; /* t returns the local time */
    SEND frame(i, s_i) **TO** *D*;
    WHILE NOT RECEIVED *Sync_Request(i,* s_i^c *)* **DO WAIT**;
    StartSchedulingFrames(s_i^c); /* server enters the regular scheduling loop */
END;

B Shifting Strategy with Different Delay Values

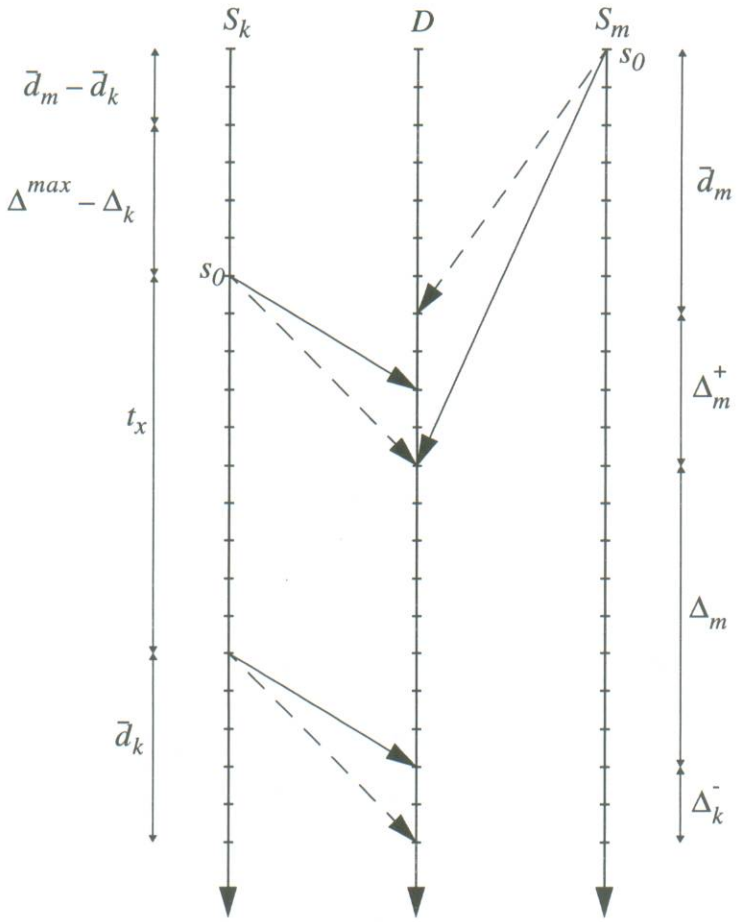

Figure 29: Worst Case Scenario for Different Jitter Values

In order to compensate for different delays when applying the start-up protocol substream k is assumed to be shifted $\bar{d}_m - \bar{d}_k$ seconds forward on time axis. According to the proof in theorem 5 t_x is calculated as follows.

$$t_x = \bar{d}_m + \Delta_m^+ + \Delta_m + \Delta_k^- - \left(\bar{d}_k + \Delta^{max} - \Delta_k + \bar{d}_m - \bar{d}_k \right)$$
$$= \Delta_m^+ + \Delta_m + \Delta_k^- - \Delta^{max} + \Delta_k$$
$$= \Delta_m^+ + \Delta_k^- + \Delta_k$$
$$= \Delta_m^+ + 2\Delta_k^- + \Delta_k^+$$

C Buffer Level Plots

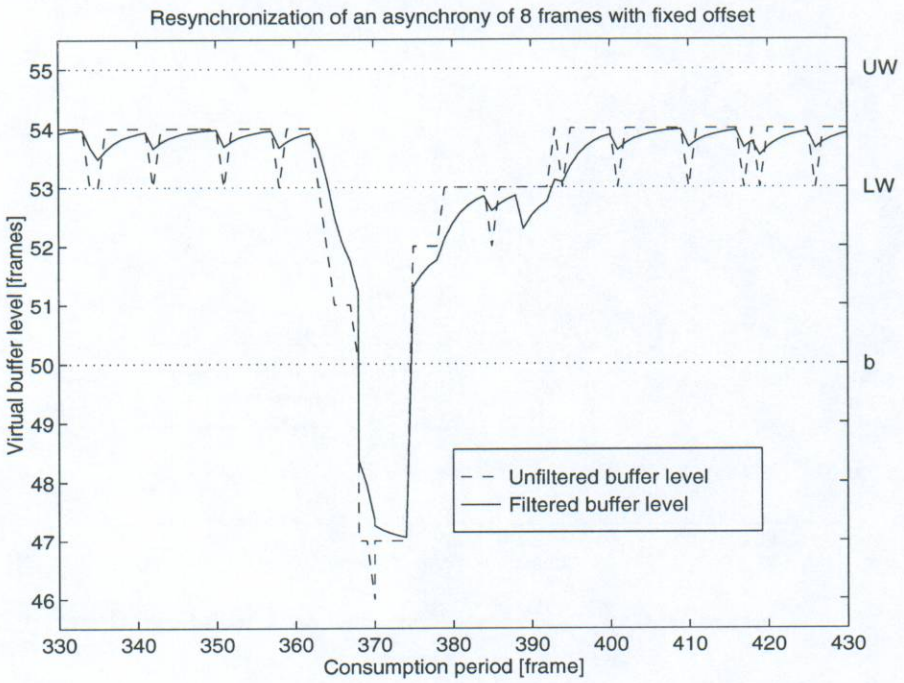

Figure 30: Gap of -8 Frames Resynchronized with Fixed Offset.

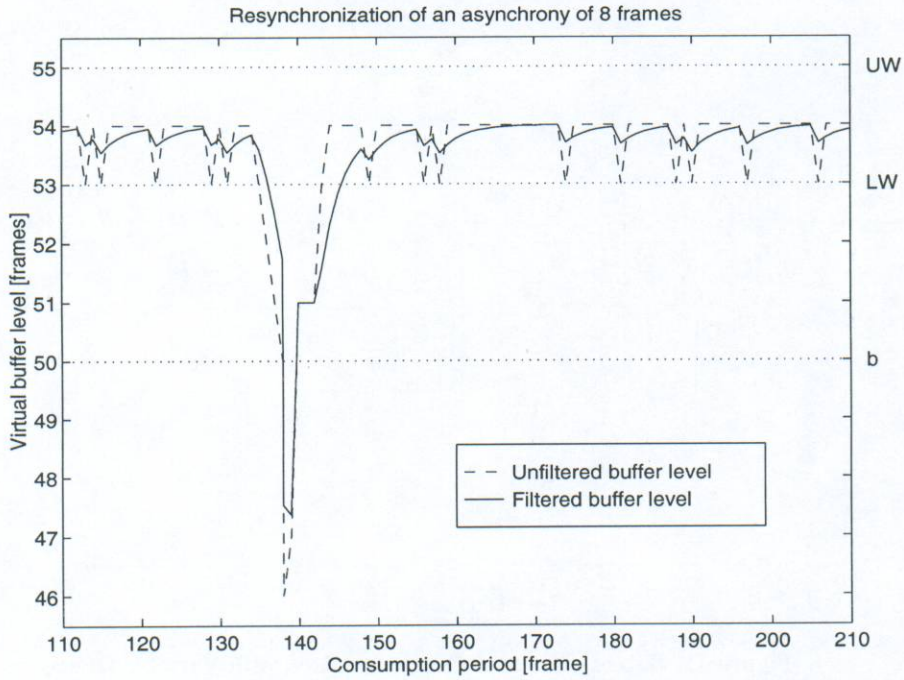

Figure 31: Gap of -8 Frames Resynchronized with Variable Offset.

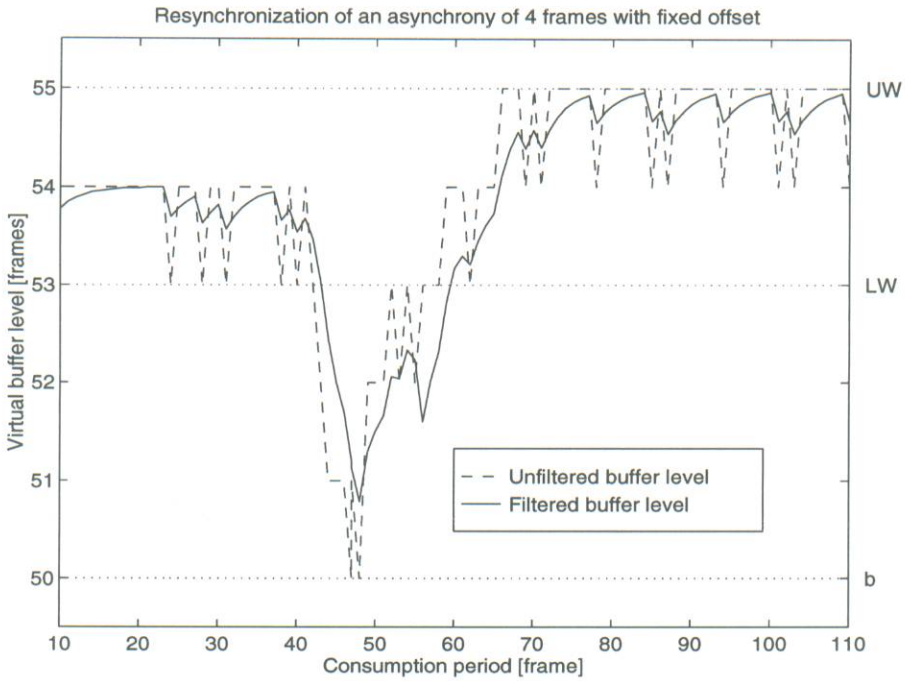

Figure 32: Gap of -4 Frames Resynchronized with Fixed Offset.

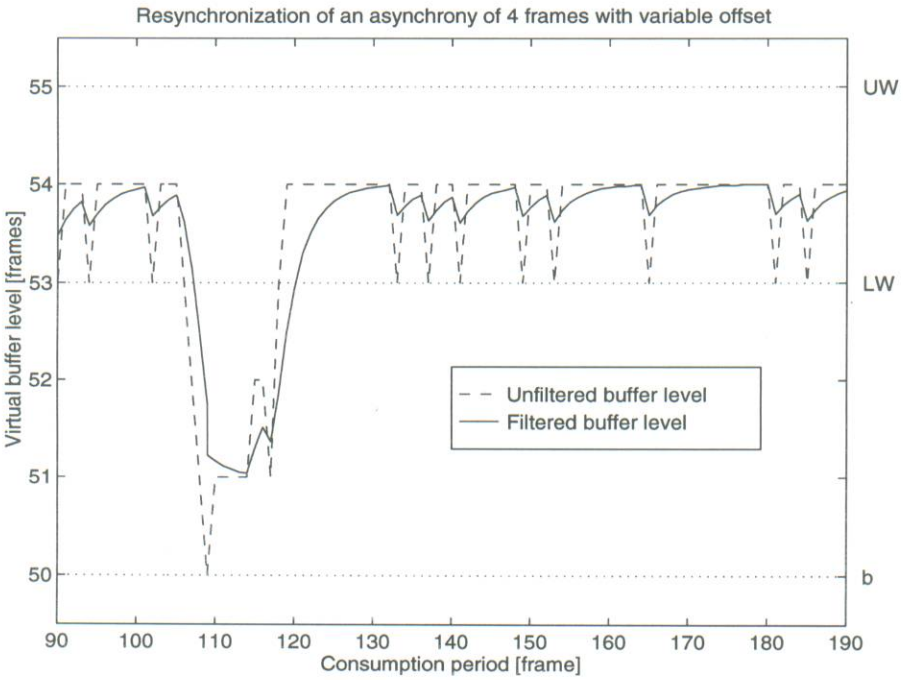

Figure 33: Gap of -4 Frames Resynchronized with Variable Offset.

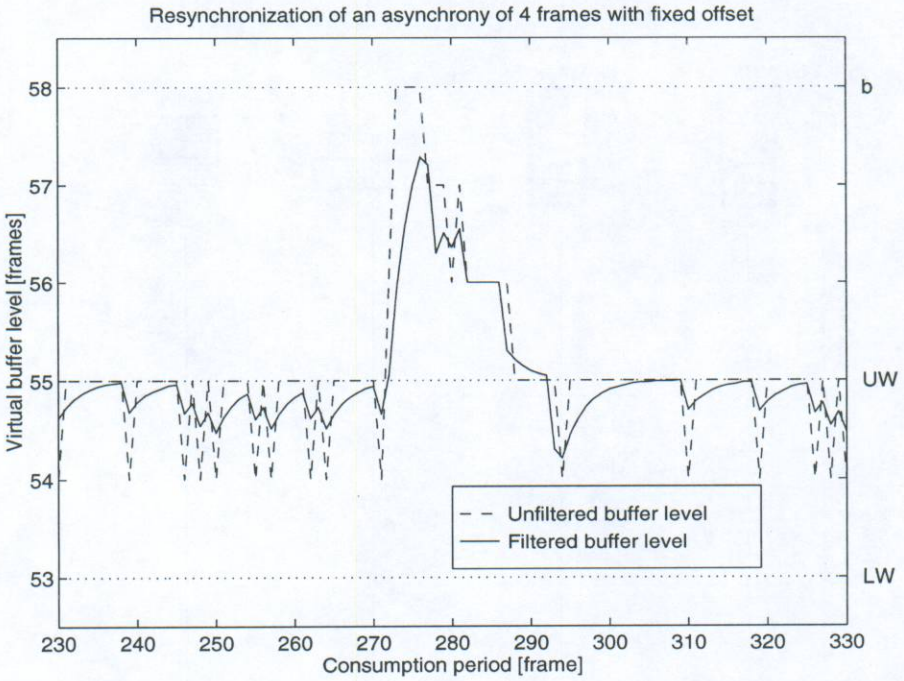

Figure 34: Concentration of +4 Frames Resynchronized with Fixed Offset.

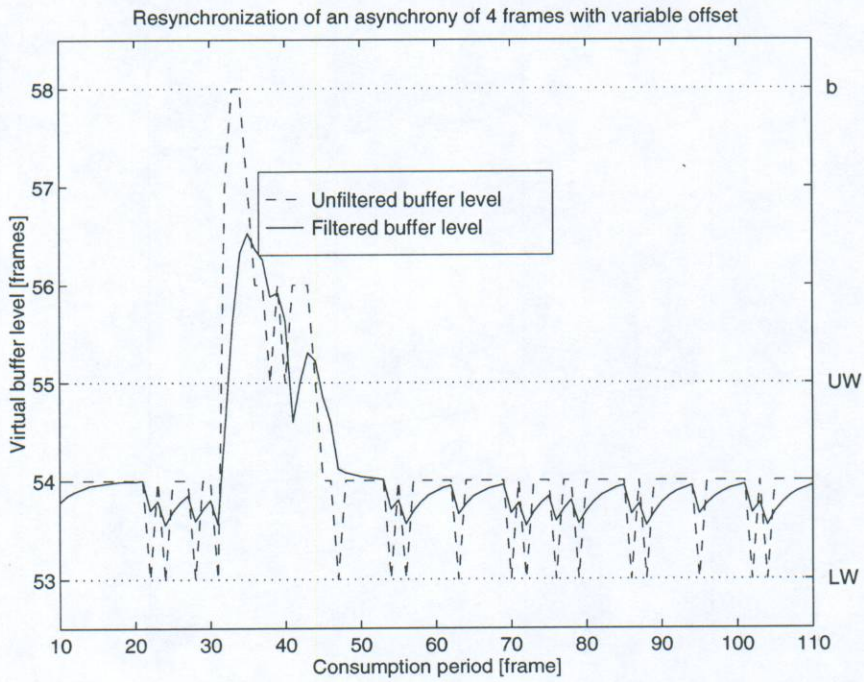

Figure 35: Concentration of +4 Frames Resynchronized with Variable Offset.

D Example for the Protocol Flow for the Retrieval of a Video

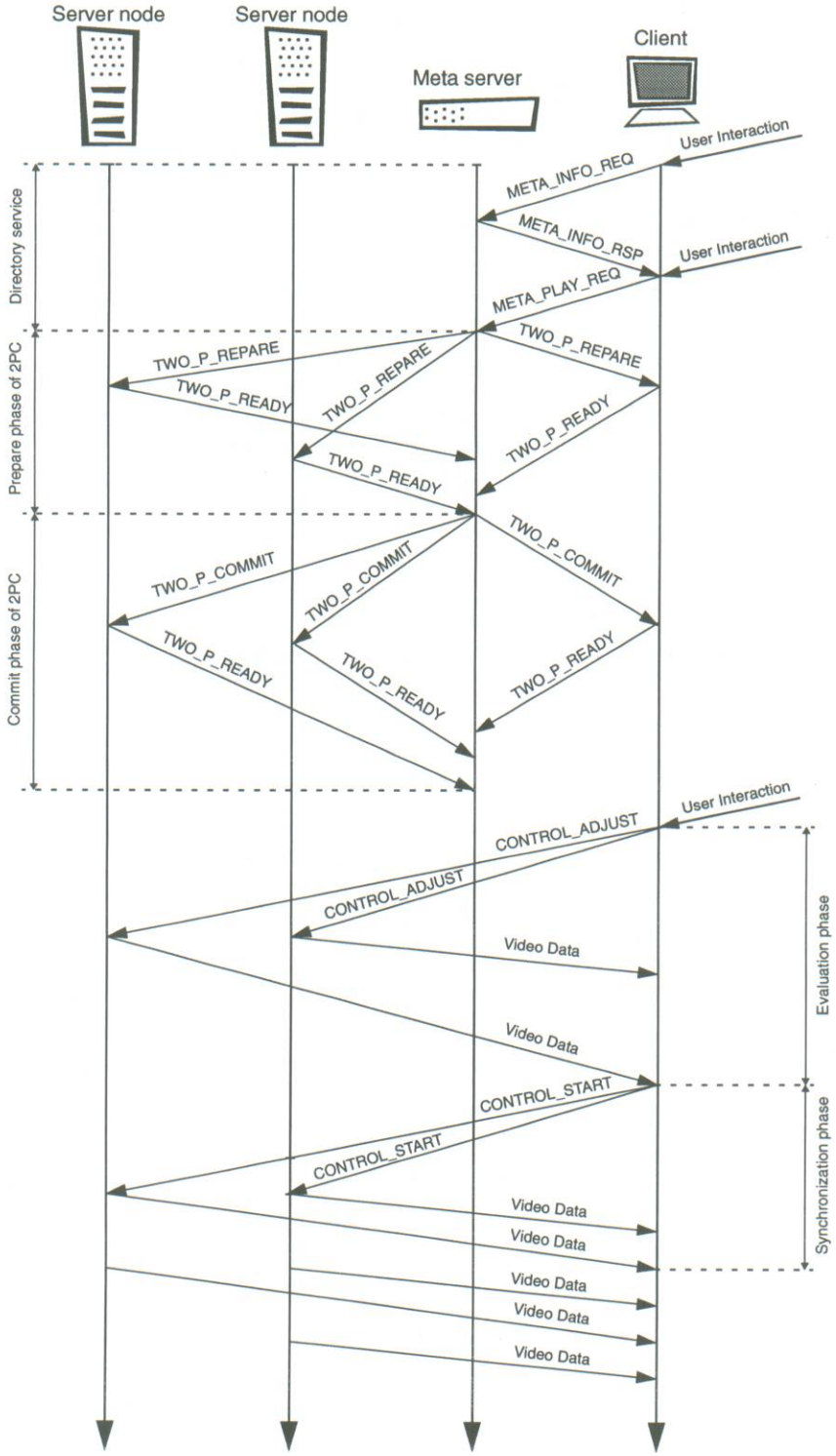

Figure 36: Protocol Flow for the Retrieval of a Video

E Structure of the Video Transmission Protocol Data Units

```
struct DataHeader {
  int           id;         // PDU identifier
  int           SID;        // unique Stream identifier
  int           FrameNum;   // unique frame number
  int           FrameSize;  // frame size in bytes
  int           PDUNum;     // number of PDU
  int           Offset;     // absolut distance (bytes) to begining of frame
  int           Length;     // amount of data included in Data[]
  long          stime;      // time stamp for sending time
  unsigned long speriod;    // scheduling period
};

struct DataPDU {
  DataHeader    Header;
  char          Data[4000];
};

struct ControlPDU {
  int           id;          // PDU identifier
  int           code;        // feedback command from client or server
  long          timestamp;   // synchronisation time, general purpose
  unsigned long speriod;     // scheduling period
  int           data;        // command related data eg. a frame number
};
```

F Structure of the Buffer Queues

```
class Buffer {
public:
  int        FrameNum;
  int        FrameLength;    // absolut length in bytes
  u_int      status;         // reserved, free, ...
  long       rTime;          // remote time, means send time
  long       lTime;          // local time, means receive time
  long       buffertime;     // time, when inserted in buffer
  char*      data;           // frame data

  Buffer ();
  ~Buffer();

  int operator<(Buffer&);
  int operator==(Buffer&);
};


class FrameQueue : public SORTLIST<Buffer> {
 public:
  Buffer* Find(int);
};
```

```
class FrameBuffer{
 public:

  FrameBuffer();
  ~FrameBuffer();

  SetBufferSize(int);

  GetFreeBuffer();
  Buffer* GetBuffer(int);
  BufferSize();

  // playable frame list sorted by framenumber
  FrameQueue       PList;

  AddFrame(Buffer*);
  RemoveFrame(int = 0);
  QueueSize();
  QueueEmpty();

  // specifys the amont of playable frames in the buffer
  int       PlayableFrames;
  int       buffer_size;

  // buffer array provides elements to be inserted in PList
  Buffer*          BList;

  // ************************************************************************
  // realization of virtual buffer management
  // ************************************************************************

  int       v_buffsize;
  int       v_bufflevel;
  float     v_filtered;
  int       v_LW;
  int       v_UW;

  v_InitBuffer(int);
  v_Reset();
  v_AddFrame();
  v_RemoveFrame();
  v_QueueSize();
  v_QueueLevel();
  v_CheckLevel();
  float v_Filter();
};


class StreamBuffer{
 public:

  FrameBuffer*          substreambuffer;
  int                   servers;
  int                   buffersize;

  StreamBuffer(int);
  ~StreamBuffer();
```

```
  Initialize(int);
  SetBufferSizes(int, int*);
//
  BufferIndex(int);

//
  GetFreeBuffer(int);
  Buffer* GetBuffer(int, int);
  Buffer* First(int = -1);
  Buffer* Next(int);
//
  AddFrame(int, Buffer*);
  RemoveFrame(int = -1, int = -1);
  QueueSize(int = -1);
  QueueEmpty(int = -1);
  BufferSize(int = -1);
};
```

G Structure of the Meta Data

```
class VideoInfo {
 public:
  int          vid;
  char         name[64];
  char         subject[64];
  char         abstract[255];
  int          fps;
  int          width;
  int          hoehe;
  int          bandwidth;
  int          qfactor;
  int          nbitspixel;
  int          number_frames;
  int          greatest_frame;
  int          smallest_frame;
  int          number_server;
  HostInfo     source[MAX_SERVER];
  int          audio_encoding;
};

struct HostInfo {
  char   hostname[32];
};
```

Bibliography

[Aga94] N. Agarwal and S. Son. "Synchronization of Distributed Multimedia Data in an Application-specific Manner." In *2nd ACM International Conference on Multimedia*, pages 141–148, San Francisco, USA, October 1994.

[Alm91] N. Almeida, J. Cabral, and A. Alves. "End-to-end Synchronization in Packet Switched Networks." In *Second International Workshop on Network and Operating System Support for Digital Audio and Video*, volume 614 of *Lecture Notes in Computer Science*, pages 84–93, Heidelberg, Germany, 1991. Springer.

[And91] D. Anderson and G. Homsy. "A Continuous Media I/O Server and Its Synchronization Mechanism." *IEEE Computer*, 24(10):51–57, 1991.

[Ber94] C. Bernhardt and E. Biersack. "Video Server Architectures: Performance and Scalability." In *Proceedings of the 4th Open Workshop on High Speed Networks*, pages 220–227, Brest, France, September 1994.

[Ber95a] C. Bernhardt and E. Biersack. "A Scalable Video Server: Architecture, Design and Implementation." In *Proceedings of the Realtime Systems Conference*, pages 63–72, Paris, France, January 1995.

[Ber95b] C. Bernhardt and E. Biersack. "The Server Array: A Novel Architecture for a Scalable Video Server." In *Proceedings of the Distributed Multimedia Conference*, pages 63–72, Stanford, USA, August 1995.

[Blu94] C. Blum. "Synchronization of Live Continuous Media Streams." In *Proceedings of the 4th Open Workshop on High-Speed Networks*, Brest, September 1994.

[Bul91] D. C. Bultermann and R. Van Liere. "Multimedia Synchronization and UNIX." In *Network and Operating System Support for Digital Audio and Video*, Lecture Notes in Computer Science, pages 108–119, Germany, November 1991. Springer.

[Cen95] S. Cen, C. Pu, R. Staehli, C. Cowan, and J. Walpole. "A Distributed real-Time MPEG Video Audio Player." In *Proceedings of the 5th International Workshop on Network and Operating System Support for Digitial Audio and Video (NOSSDAV'95)*, Durham, NH, April 1995.

[Cha94] A. Chakrabati and R. Wang. "Adaptive Control for Packet Video." In *Proceedings of the International Conference on Multimedia Computing*, pages 56–62, May 1994.

[Cor92] J. S. Cormac. "Synchronisation Services for Digital Continuous Media." Master's thesis, University of Camebridge, Camebridge, England, October 1992. Good overview.

[Cor95] J. S. Cormac. "Position Statement: To Use or Avoid Global Clocks." IEEE Workshop on Multimedia Synchronization (Sync'95), Mai 1995.

[Del94] D. Deloddere, W. Verbiest, and H. Verhille. "Interactive Video On Demand." *IEEE Communications Magazine*, 32(5):82–88, May 1994.

[Den94] J. Dengler and W. Geyer. "A Video Server Architecture Based on Server Arrays." Technical report, Institut Eurecom, Sophia-Antipolis, France, October 1994.

[Den95] J. Dengler. "Admission Control Strategies and Design Issues for Video Servers." Master's thesis, University of Mannheim, Mannheim, November 1995.

[Eff93] 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.

[Ehl94] L. Ehley, B. Furht, and M. Ilyas. "Evaluation of Multimedia Synchronization Techniques." In *International Conference on Multimedia Computing and Systems*, pages 514–519, Boston, Massachusetts, Mai 1994. IEEE.

[Esc94] J. Escobar, C. Patridge, and D. Deutsch. "Flow Synchronization Protocol." In *ACM Transactions on Networking*, volume 2, pages 111–121. IEEE, April 1994.

[Fed94] C. Federighi and L. A. Rowe. "A Distributed Hierarchical Storage Manager for a Video-on-Demand System." In *Proceedings of IS&T/SPIE Symposium on Electronical Imaging Science & Technology, Storage and Retrieval for Image and Video Databases II*, San Jose, CA, February 1994.

[Fur94] B. Furht. "Multimedia Systems: An Overview." *IEEE Multimedia*, pages 47–59, Spring 1994.

[Gem95] D. J. Gemmell, H. M. Vin, D. D. Kandlur, P. V. Rangan, and L. A. Rowe. "Multimedia Storage Servers: A Tutorial and Survey." *IEEE Computer*, 28(5):40–49, May 1995.

[Ish95] 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.

[Ker90] B. W. Kernighan and D. M. Ritchie. *Programmieren in C*. Carl Hanser Verlag, Munich, second edition edition, 1990.

[Kni94] E. W. Knightly, R. Mines, and H. Zhang. "Determinstic Characterization and Network Utilizations for Several Distributed Real-time Applications." In *Proceedings of IEEE WORDS'94*, Dana Point, CA, October 1994.

[Koe94] D. Koehler and H. Mueller. "Multimedia Playout Synchronization Using Buffer Level Control." In *2nd International Workshop on Advanced Teleservices and High-Speed Communication Architectures*, Heidelberg, Germany, September 1994.

[Lai94] S. Laird and J. Youngman. *Video Development Environment - Reference Guide*. Parallax Graphics Inc., Santa Clara, CA, 1994.

[Li94] L. Li and D. Georganas. "MPEG-2 Coded- and Uncoded- Stream Synchronization Control for Real-time Multimedia Transmission and Presentation over B-ISDN." In *2nd ACM International Conference on Multimedia*, pages 239–245, San Francisco, USA, October 1994.

[Lit90] T. Little and A. Ghafoor. "Synchronization and Storage Models for Multimedia Objects." *IEEE Journal on Selected Areas in Communications*, 8(3):413–427, April 1990.

[Lit91] T. Little, A. Ghafoor, C. Chang, and P. Berra. "Multimedia Synchronization." *IEEE Data Engineering Bulletin*, 14(3):26–35, September 1991.

[Lit92] 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.

[Lit94] T. D. C. Little and D. Venkatesh. "Client-Server Metadata Management for the Delivery of Movies in a Video-On-Demand System." Technical Report 12-14-1994, Multimedia Communications Laboratory, Department of Electrical, Computer and Systems Engineering, Boston University, Boston, MA, 1994.

[Mas90] H. Massalin and C. Pu. "Fine-Grain Adaptive Scheduling Using Feedback." *Computing System*, 3(1):139–173, 1990.

[Mat92] The MathWorks, Inc., Natick, MA. *Matlab User's Guide*, student edition edition, 1992. Published by Prentice-Hall, Inc.

[Mil91] D. Mills. "Internet Time Synchronization: The Network Protocol." *IEEE Transactions on Communications*, 39(10):1482–1493, October 1991.

[Ous94] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Massachusetts, 1994.

[Ran92] P. V. Rangan, H. M. Vin, and S. Ramanathan. "Designing an On-Demand Multimedia Service." *IEEE Communications Magazine*, 30(7):56–65, July 1992.

[Ran93] P. Rangan, S. Ramanathan, H. M. Vin, and T. Kaepppner. "Techniques for Multimedia Synchronization in Network File Systems." *Computer Communications*, 16(3):168–176, March 1993.

[Rot95a] K. Rothermel. "Position Statement: State-of-the-Art and Future Research in Stream Synchronization." IEEE Workshop on Multimedia Synchronization (Sync'95), Mai 1995.

[Rot95b] K. Rothermel and T. Helbig. "An Adaptive Stream Synchronization Protocol." In *5th International Workshop on Network and Operating System Support for Digital Audio and Video*, Durham, New Hampshire, USA, April 1995.

[Rot95c] K. Rothermel, T. Helbig, and S. Noureddine. "Activation Set: An Abstraction for Accessing Periodic Data Streams." In *Multimedia Computing and Networking*, volume 2417, San Jose, California, February 1995. IS&T/SPIE.

[Row94] L. A. Rowe, J. S. Boreczky, and C. A. Eads. "Indexes for User Access tp Large Video Databases." In *Proc. of IS&T/SPIE 1994 International Symposium on Electronic Imaging: Science and Technology*, San Jose, CA, February 1994.

[San93] 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.

[Sch94] M. Schader and S. Kuhlins. *Programmieren in C++*. Springer-Verlag, Heidelberg, second edition edition, 1994.

[Ste90a] R. Steinmetz. "Synchronization Properties in Multimedia Systems." *IEEE Journal On Selected Areas in Communications*, 8(3):408–412, April 1990.

[Ste90b] W. R. Stevens. *UNIX Network Programming*. Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[Ste93a] R. Steinmetz. *Multimedia-Technologie*. Springer Verlag, Heidelberg, Germany, 1993.

[Ste93b] R. Steinmetz and C. Engler. "Human Perception of Media Synchronization." Technical Report 43.9310, IBM European Networking Center, Vangerowstrasse 18, 69020 Heidelberg, Germany, 1993.

[Ste95] R. Steinmetz. "Analyzing the Multimedia Operating System." *IEEE Multimedia*, 2(1):68–84, Spring 1995.