Multimedia Systems (1997) 5:324–336

Multimedia Systems

An adaptive protocol for synchronizing media streams

Kurt Rothermel, Tobias Helbig[1]

University of Stuttgart, Institute of Parallel and Distributed High-Performance Systems (IPVR), Breitwiesenstrasse 20-22, D-70565 Stuttgart, Germany
e-mail: {rothermel,helbig}@informatik.uni-stuttgart.de

Abstract. Stream synchronization is widely regarded as a fundamental problem in the field of multimedia systems. Solutions to this problem can be divided into adaptive and rigid mechanisms. While rigid mechanisms are based on worst case assumptions, adaptive ones monitor the underlying network and are able to adapt themselves to changing network conditions. In this paper, we will present an adaptive stream synchronization protocol. This protocol supports any kind of distribution of the sources and sinks of the streams to be synchronized. It is based on a buffer-level control mechanism, allowing immediate corrections when the danger of a buffer overflow or underflow is recognized. Moreover, the proposed protocol is flexible enough to support a wide variety of synchronization policies, which can be dynamically changed while synchronization is in progress. Finally, the message overhead of this protocol is low, because control messages are only exchanged when network conditions change.

Key words: Distributed systems – Communication networks – Multimedia – Stream synchronization – Quality of service

1 Introduction

The evolution of broadband networks and multimedia technologies have significantly contributed to the emergence of new multimedia applications, integrating various media types, such as text, graphics, audio and video. These data typically possess timeliness requirements with respect to their presentation. Media synchronization mechanisms are needed to assure the correct temporal alignment of such time-critical activities.

Media synchronization can be divided into event-based synchronization and stream (or continuous) synchronization. While event-based synchronization refers to synchronization activities performed in response to events such as user interaction, stream synchronization is an on-going commitment to a repetitive pattern of event-based synchronization relationships, such as a 'lip sync' relationship between the individual data units in an audio and video stream (Campell et al. 1992). The stream synchronization can be further subdivided into intra-stream synchronization and inter-stream synchronization. While the former refers to preserving temporal relationships of data within a stream, the latter deals with the temporal dependencies across streams.

Intra-stream synchronization is concerned with a single stream. A source of a stream produces data units and transmits them over a transmission path to one or more sinks. The transmission path inevitably introduces some variation in the delay of each delivered data unit, which traditionally has been called jitter. Intra-stream synchronization requires the jitter to be removed before playing out the data units, which is done by buffering the incoming data. A data unit is rendered at a designated play-out point, and is buffered if it arrives before this point. Data arriving after the associated play-out point is useless in reconstructing the corresponding real-time signal.

Multimedia applications have been classified into adaptive and rigid applications (Clark et al. 1992). The latter class of applications use an *a priori* transfer delay bound advertised by the underlying network to set the play-out point. The play-out point is kept fixed regardless of the actual delay experienced. In contrast, for adaptive applications, the sink measures the transfer delay experienced by arriving data units and then adaptively moves the play-out point to the minimum delay that still produces a sufficiently low loss rate.

Rigid applications are typically based on a so-called guaranteed (or deterministic) service (Ferrari 1990a, b), whose service commitment is based on a worst case analysis. Adaptive applications will generally have an earlier play-out point than rigid applications, and hence will have a shorter end-to-end delay. This is because the application's estimate of the *post facto* bound on actual delay will likely be less than the *a priori bound* pre-computed by the underlying network (Clark et al. 1992). On the other hand, the loss rate of adaptive applications is likely to be higher, as they depend on the assumption that the transfer delay in the near future will be "similar" to the one in the recent past. Any viola-

[1] *Contact address*: Philips Research Laboratories, Weisshausstrasse 2, D-52066 Aachen, Germany; helbig@pfa.research.philips.com
Correspondence to: T. Helbig

tion of this assumption in the direction of increased delays may cause data units missing the play-out point. Though the application will then immediately adapt the play-out point accordingly, it may momentarily experience data loss. Note that the notion of "similar" leaves room for tuning adaptive protocols. The more "similar" delays may differ, the more data has to be buffered and the bigger is the end-to-end delay.

There is a need for both classes of applications. Applications that cannot tolerate any service interruption, such as a remote surveillance system or tele-medicine, will be typically rigid. On the other hand, if the application performance is sensitive to the end-to-end delay and a briefly degraded quality is tolerable, then the application should be adaptive. For example, end-to-end delay is crucial in most CSCW applications, because there is often real-time interaction between the participants of a session. For many of those applications, a short end-to-end delay is more important than a perfect data delivery. They often can tolerate the loss of a certain fraction of data units with only a minimum distortion of the real-time signal.

Inter-stream synchronization determines the play-out points for a group of data streams, based on the temporal relationships existing between the group members. To ensure that a stream group is played out synchronously, temporally related data units are to be associated with the same play-out point. Adaptive inter-stream synchronization protocols monitor the actual transfer delay of each of the group's streams and are able to synchronously adapt the play-out point for every group member to reflect changes in network conditions. In this paper, we will present an *adaptive protocol for inter-stream and intra-stream synchronization*. This protocol, called Adaptive Synchronization Protocol (ASP), has the following major characteristics:

- **Distributed sources and sinks**
  ASP supports any kind of distribution of the group of streams to be synchronized. The streams of a group may originate from sources residing on different nodes and may be played out at sinks located at various nodes. The individual streams may be point-to-point or point-to-multipoint.

- **Immediate reactions on changing network conditions**
  ASP monitors the actual transfer delay indirectly by means of a buffer control mechanism and adapts the play-out point only when a stream becomes critical. A stream is defined to be critical if it runs the risk of a buffer underflow or overflow. A nice property of our algorithm is that each stream may immediately adapt its play-out point when it becomes critical. Allowing streams to react immediately in critical situations may decrease the loss rate significantly.

- **Low message overhead**
  ASP only exchanges control messages when adaptions are to be performed due to changing network conditions or quality of service requirements. Consequently, there is basically no message overhead if network conditions and QoS requirements are rather stable over time. The significant reduction of the message overhead for synchronizing streams is achieved by making the transition from a periodic exchange of the streams' state information to reacting on changing conditions only.

- **Flexibility**
  ASP is a flexible mechanism that can form the base for various synchronization policies, such as a "minimum delay" and "minimum loss" policy. It allows an application to dynamically adjust the quality of service perceived by an end-user. In particular, an application can individually adjust protocol parameters to achieve the desired trade-off between end-to-end delay and data loss rate and can modify these parameters even while synchronization is in progress.

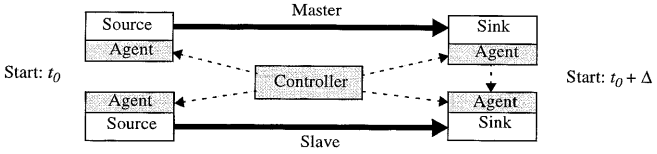


Fig. 1. System model

The remainder of the paper is structured as follows. After introducing the basic principles of ASP in Sect. 2, the actual synchronization mechanism is described in Sect. 3. The proposed mechanism can be adapted to various application needs and forms the basis for different synchronization policies. This is discussed in Sect. 4. The stability aspects and buffer requirements are treated in Sect. 5. A discussion of simulation results and performance measurements is given in Sect. 6. The paper concludes with a discussion of related work and a summary.

2 Basic principles and concepts

The existence of synchronized clocks not only simplifies media synchronization significantly but also allows for more efficient solutions. Some of the protocols based on synchronized clocks use global time only for the timing of control operations, such as starting, stopping or adjusting a group of streams at the same point in global time (e.g., see Campell et al. 1992). Others additionally use global time as the temporal basis for scheduling the play-out of data units (e.g., see Escobar et al. 1994). In this section, we will introduce the basic principles of the latter class of synchronization protocols. Before, however, we have to introduce some terminology.

The set of streams which are to be played out in a synchronized fashion is called *synchronization group* (or sync group for short). For each sync group, there exist a single synchronization *controller* and several *agents* (see Fig. 1). The controller is a software entity that maintains state information and performs control operations concerning the entire sync group. In particular, it controls the start-up procedure, and enforces the synchronization policy chosen by the user. The controller communicates with the agents, which are software entities controlling individual streams. For each stream there exist a sink agent and a source agent, which commonly realize the functionality for starting and stopping the stream, as well as modifying the stream's play-out rate. Sink agents

may communicate with each other in order to adapt play-out points.

We are considering continuous data streams, which may originate from live or stored media sources. For the sake of simplicity, we will assume relative timestamping, i.e., the timestamp of a stream's first data unit is zero, and all succeeding data units are timestamped relative to time zero.

The basic principle of stream synchronization adopted by ASP and other protocols exploiting synchronized clocks is fairly simple. All source agents in the sync group start sending data units at the same time, say t_0. A data unit u is sent at time t_0 + TS(u), where TS(u) denotes the timestamp associated with u. Each sink in the sync group starts the presentation of its stream at time $t_0 + \Delta$. Each data unit u is played out at time $t_0 + \Delta$ + TS(u), which is u's play-out point. Clearly, Δ must be big enough to allow at least the first data unit of each stream to arrive at its sink by time $t_0+\Delta$. Roughly speaking, Δ determines the end-to-end delay of a sync group: $\Delta = max(d_i : i$ in sync group), where d_i denotes the delay of stream i. Since different streams may have different transfer delays, buffering is required at the sink sites. Data units arriving before $t_0 + \Delta$ are buffered, which means that different transfer delays are equalized by means of buffering. This principle is typically used for the synchronization of live streams, but may also be applied to the retrieval of stored data.

In the case of non-adaptive protocols, Δ is determined during protocol initialization and then it is fixed afterwards. Note, this approach implies that worst case assumptions are made about stream delays, which results in a worst case end-to-end delay for the sync group, independent of the actual delays. If Δ is fixed, the synchronization mechanism is trivial. All that has to be done is to start the transfer and the presentation of the streams in the way described above. Once started, the streams remain in sync because play-out times are derived from global time, i.e., no control messages have to be transferred after initialization. The message overhead caused by the underlying clock synchronization mechanism is amortized among all applications making use of synchronized clocks.

With adaptive protocols, Δ is based on the actual stream delays rather than worst case assumptions. Stream delays are monitored and Δ is adapted in response to delay changes. Moreover, the quality of service (QoS) can be changed dynamically. By increasing Δ, the probability of data loss due to late arrival of data units is decreased, whereas the end-to-end delay is increased. Conversely, decreasing Δ increases the loss probability and decreases the end-to-end delay. Adaptive protocols are a bit more complex than non-adaptive ones. In addition to deriving a common Δ for the streams to be synchronized, adaptive protocols need to have functions for controlling the adaption process, which may be distributed over several sink sites. Those functions monitor stream delays, react on changing QoS demands, and trigger adaptions as needed. Of course, adaptions have to be performed in a coordinated fashion to preserve synchronization. In particular, all streams in a sync group have to agree on a new Δ value and switch to it without losing synchronization.

In ASP, adaptions are coordinated by a *dynamic master/slave algorithm*. Each sink agent monitors the transfer delay by controlling the stream's play-out buffer. During normal operation, there is one stream responsible for adapting Δ, the so-called *master stream*. The master's decision of when and how to adapt is entirely based on its local monitoring. Whenever the master's sink agent decides to change Δ, it propagates its decision to the sink agents of all the other streams in the sync group, the so-called *slave streams*. The algorithm is dynamic in the sense that, whenever a slave stream becomes critical, it may immediately become a master and perform the appropriate adaptions. Obviously, with this algorithm it may happen that there exist multiple masters at the same time. Our protocol is able to handle those situations without losing synchronization and ensures that after a certain recovery period the sync group ends up with a single master stream.

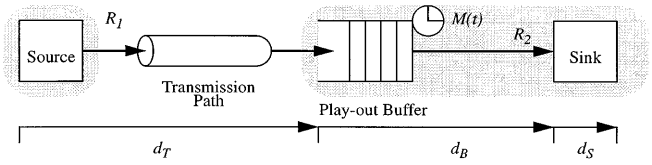


Fig. 2. Data stream and delay model

Our model of stream transmission and buffering is depicted in Fig. 2. The data units of a stream are produced by a source with a *nominal rate* R_1 and are transmitted to one or more sinks over a unidirectional transmission path. We will use a transmission path as an end-to-end abstraction describing the flow of data between end-points of applications. In this sense, a transmission path may be a communication channel (e.g., a transport connection) directly linking a source with a set of sinks, or it may represent a sequence of processing elements, such as codecs, mixers or filters, connected with each other by communication channels. Before the data units are played out, they are stored in a *play-out buffer* at the sink's site. From this buffer, data units are released with a *release rate* R_2.

With ASP, Δ is modified by increasing or decreasing release rate R_2 for a certain amount of time. During normal operation R_2 equals R_1. In order to increase Δ, ASP decreases R_2 for a period of time, causing an increase in buffer delay. Conversely, increasing R_2 results in a decrease of Δ. Sinks must be able to adapt to changing release rates. Either a sink can adapt its consumption rate accordingly, or adaptions are achieved by means of skipping or duplicating data units (Anderson and Homsy 1991). Also media-specific methods are conceivable, such as adjusting silent periods in voice data streams.

On its way from generation to play-out, a data unit is delayed at several stages. It takes a data unit a *transfer delay* d_T until it arrives in the buffer at the sink's site. This includes all the times for generation, communication, processing, as well as the transfer into the buffer. In the buffer, a data unit is delayed by a *buffering delay* d_B before it is delivered to the sink device. In the sink, a data unit experiences a *play-out delay* d_S before it is actually presented. The time from the generation to the presentation is the *end-to-end delay*.

The *media time* $M(t)$ specifies the stream's temporal state of play-out. It is derived from timestamp TS of the data unit that is next to be released from the play-out buffer: $M(t)$ = TS $- d_S$. However, the granularity of media time

would be too coarse if it were simply based on timestamps. Therefore, media time is interpolated between timestamps of data units to achieve the required granularity.

We will assume that control messages are communicated reliably. The required level of reliability is typically provided by virtual circuits or reliable datagrams. Further, it is assumed that the system clocks of the nodes participating in a sync group are approximately synchronized to within ε of each other, i.e., no clock value differs from any other by more than ε. Well-established protocols, such as the Network Time Protocol (Mills 1990), achieve clock synchronization with ε in the lower milliseconds range.

3 The adaptive synchronization protocol

This section presents the Adaptive Synchronization Protocol (ASP), which can be separated into four rather independent subprotocols. After a brief overview, we will describe each of these protocols in detail. It is important to mention, that this section concentrates on mechanisms, while possible policies exploiting these mechanisms will be discussed in the next section.

3.1 Overview of the protocols

ASP consists of the following four subprotocols: the start-up protocol, buffer control protocol, master/slave synchronization protocol, and master switching protocol. *The start-up protocol* initiates the data transmission at the sources and the play-out process at the sinks. Start-up is coordinated by the controller, which derives start-up times from estimated transmission times, selects an initial master stream depending on the chosen synchronization policy and sends control messages containing the start-up times to the agents.

The *buffer control protocol* is a purely local mechanism, performed by the master stream's sink agent to keep the play-out buffer delay in a given target area. The determination of the target area depends on the applied synchronization policy, and thus is not subject to this mechanism itself. Whenever the buffer delay moves out of the given target area, the buffer control protocol regulates the master's release rate accordingly. It is this protocol that adjusts the play-out point of the master stream when network conditions or QoS requirements change.

The *master/slave synchronization protocol* is initiated whenever the master stream's release rate is adjusted by the above protocol. To ensures inter-stream synchronization, the sink agent of the master stream propagates an appropriate specification of this adjustment to the sink agents of all slave streams. Upon receipt of this information, an agent adjusts the release rate of its slave stream accordingly. It is this protocol that makes sure that play-out points are adjusted consistently across all streams in the sync group.

The *master switching protocol* allows to switch the master role from one stream to another at any point in time. The protocol involves the sink agents and the controller, which is responsible for granting the master role. Switching the master role becomes necessary when some slave stream enters the critical state. A critical slave becomes a so-called tentative master, whose release rate can be adjusted immediately. The protocol takes care of the fact that there may be a master and several tentative masters at the same point in time and makes sure that the sync group eventually ends up with a single master.

3.2 Start-up protocol

Our start-up procedure is very similar to that described in Escobar et al. 1994. The controller initializes the synchronous start-up of a sync group's data streams by sending *Start* messages to each sink and source agent. Each *Start* message contains besides other information a start-up time. All source agents receive the same start-up time, at which they are supposed to start transmitting data units. Similarly, all sink agents receive the same start-up time, which tells them when to start the play-out process.

Starting agents simultaneously requires the *Start* messages to arrive early enough. The start-up time t_0 of sources is derived from the current time t_{now}, the transfer delay d_m experienced by *Start* messages, and processing delays d_{proc} at the controller site: $t_0 = t_{now} + d_m + d_{proc}$. Start-up of sinks is deferred by an additional time Δ to allow the stream data to arrive at the sinks' locations and to preload buffers. This extra delay is computed from the streams' transfer delays and delays caused by buffer preloading: $\Delta = max((d_i + LWM_i) : i$ in sync group), where d_i and LWM_i denotes stream i's transfer delay and buffer delay, respectively. LWM_i mainly depends on i's jitter (for detail see next section). We assume some infrastructure component that provides access to the (estimated) jitter and delay parameters.

A *Start* message sent to a source agent contains the start time t_0 and the nominal stream rate R_N. A source agent receiving such a message starts transmission at time t_0 with rate $R_1 = R_N$. *Start* received by a sink agent includes start time $t_0+\Delta$, R_N and a flag indicating the receiver's initial role (i.e., master or slave). Furthermore, it includes some initial parameters concerning the play-out buffer (see below). A sink agent starts the play-out process at the specified time with rate $R_2 = R_N$.

Each agent starts stream transmission or play-out at the received start-up time. Therefore, the start-up asynchronicity is bounded by the inaccuracy of clock synchronization, provided *Start* messages arrive in time. However, even if some *Start* messages are too late, ASP is able to immediately resynchronize the 'late' streams.

3.3 Buffer control protocol

Before describing the protocol, we will take a closer look at the play-out buffer. The parameter $d_B(t)$ denotes the smoothed buffer delay at time t. The buffer delay at a given point in time is determined by the amount of buffered data and the rate of the stream. In order to filter out short-term fluctuations caused by jitter, some smoothing function is to be applied. ASP does not require a distinct smoothing function. Some examples are the geometric weighting smoothing function (Postel 1981): $d_B(t_i) = \alpha \cdot d_B(t_{i-1}) + (1 - \alpha) \cdot$

$ActBufferDelay(t)$, or the Finite Impulse Response Filter as used in Koehler and Müller (1994).

In ASP, all buffer-related values are measured in time units rather than bytes. A buffer of size n seconds can hold up to n seconds of the corresponding data stream. The advantage of using a temporal dimension is that the ASP mechanism becomes totally independent of the media streams to be synchronized and their encodings. Mapping the temporal size of a buffer to its size in bytes is straight-forward for CBR streams. For VBR streams, this mapping is more complicated for a number of reasons. Note that this type of mapping is needed wherever buffer space and bandwidth is to be allocated for streams. Thus, it should be provided by resource management protocols. ASP is kept independent from this mapping leading to a clear separation of stream control and resource management.

For each play-out buffer a *low-water mark (LWM)* and *high-water mark (HWM)* is defined. When $d_B(t)$ falls under *LWM* or exceeds *HWM*, there is the risk of underflow or overflow, respectively. Therefore, we will call the buffer areas below *LWM* and above *HWM* the *critical buffer regions*. As will be seen below, ASP takes immediate corrective measures when $d_B(t)$ moves into either one of the critical buffer regions. Note that the quality of intra-stream synchronization is primarily determined by *LWM* and *HWM* values. The buffer parameters are set by the ASP client according to application and network characteristics (see Sect. 4).

The buffer control protocol is executed locally at the sink site of the master stream. Its only purpose is to keep $d_B(t)$ of the master stream in a so-called *target area*, which is defined by an *upper target boundary (UTB)* and a *lower target boundary (LTB)*. While the high- and low-water marks describe the intervention marks that cause a slave stream's reactions to avoid the overflow and underflow of its buffer, the target area causes the master stream to follow changes in transfer delays. Hence, the role of the stream determines the marks used for reactions. Clearly, the target area must not overlap with a critical buffer region. The location and width of the target area is primarily determined by the chosen synchronization policy (see Sect. 4). For example, to minimize the overall delay the target should be close to *LWM*.

The buffer delay $d_B(t)$ may float freely between the lower and upper target boundary without triggering any rate adaptions. Changing transmission delays (or a modification of the target area requested by the controller) may cause $d_B(t)$ to move out of the target area. When this happens, the master enters a so-called *adaption phase*, whose purpose is to move $d_B(t)$ back into the target area.

At the beginning of the adaption phase, the release rate is modified accordingly. The adapted release rate is $R_2^A = R_N \cdot (1 + R_{corr})$, where $R_{corr} = (d_B(t) - (LTB + (UTB - LTB)/2))/L$ is the relative correction rate. Length L of the adaption phase determines how aggressive the algorithm reacts: the smaller L, the more aggressive the algorithm. At the end of the adaption phase, it is checked whether $d_B(t)$ has moved back into the target area. If this is the case, then R_2 is set back to R_N, otherwise another adaption phase is started.

In order to keep the slave streams in sync, each adaption of the master stream has to be propagated to the slave streams. This is achieved by the protocol described next.

3.4 Master/slave synchronization protocol

The master/slave synchronization protocol ensures that the slave streams are played out in sync with their master stream. This protocol is initialized whenever the master (or a tentative master, as will be seen in the next section) modifies its release rate. Protocol processing only involves sink agents, each of which acts either as master or slave.

Whenever the master enters an adaption phase, it performs the following operations. First, it computes the so-called target media time for this adaption phase, which is defined to be the media time the master stream will reach at the end of this phase. Assume that the adaption phase starts at real-time t_s and is of length L. Then the target media time is $M(t_s + L) = M(t_s) + L \cdot R_2^A$. Subsequently, the master propagates an *Adapt* message to each slave in the sync group. An *Adapt* message includes the following information: $(TS, t_e, M(t_e))$, where $t_e = t_s + L$ is the time the adaption phase ends, $M(t_e)$ specifies the media time at the end of the adaption phase, and TS is a structured timestamp for ordering competing *Adapt* messages.

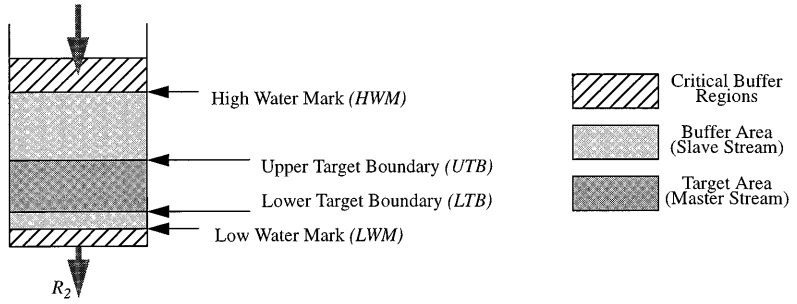
When a slave receives an *Adapt* message, it immediately enters the adaption phase by modifying its release rate according to the received target media time (see Fig. 5). The modified release rate $R_2^A = R_N \cdot (M(t_e) - M(t_a))/(t_e - t_a)$, where t_a denotes the time at which the slave received *Adapt*. At time t_e (i.e., at the end of the adaption phase), R_2 is set back to R_N.

Obviously, this protocol ensures that at the end of each adaption phase all streams in the sync group reach the same target media time at the same point in real-time. Between two adaption phases, streams stay in sync as their nominal release rates are derived from global time.

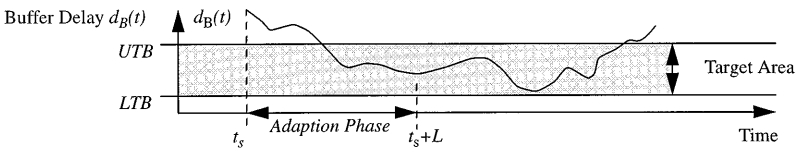
As with all synchronization schemes based on the notion of global time, skew among sinks is introduced by the inaccuracy of synchronized clocks, which is assumed to be bounded by ε. In our protocol, an additional source of skew is the adaption of release rates at different points in time. The worst case skew S_{max} during the adaption phase of the master depends on transfer time d_m of the *Adapt* message and the master's relative correction rate R_{corr} : $Skew_{max} = d_m \cdot |R_{corr}| + \varepsilon$, where the term $d_m \cdot |R_{corr}|$ denotes the skew caused by the delay of the *Adapt* messages. Our simulation results in Sect. 6 will show that the value of this term typically is in the range of 10–15 ms in wide area networks. If no adaption is in progress, the skew is bounded by ε.

With a slight modification of our protocol, we can achieve a skew bound of ε even during the adaption phase. We only have to make sure that the master and its slaves enter the adaption phase at the same point in global time. Assume that the master's buffer delay moves out of the target area at time t. Instead of entering immediately the adaption phase, it only sends out *Adapt* messages to all of its slaves, while the start of the actual adaption phase is deferred by some time δ. An *Adapt* message contains the following parameters $(TS, t_s, t_e, M(t_e))$, where the additional parameter $t_s = t + \delta$ denotes the starting time of the adaption phase. All other parameters have the same semantics as above.

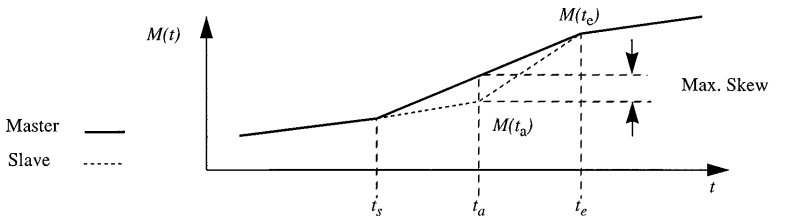
A slave receiving an *Adapt* message checks whether it received this message later than t_s. If this is the case, the



3



4



5

Fig. 3. Buffer regions and intervention marks of the play-out buffer

Fig. 4. Buffer delay adaption

Fig. 5. Master/slave synchronization

slave immediately enters the adaption phase. Otherwise, it waits for entering this phase until time t_s is reached. Obviously, if δ is set to the maximum delay of control messages, the master and all of its slaves start the adaption at the same point in global time. Now the potential inaccuracy of the synchronized clocks is the only source of skew, i.e., $Skew_{max} = \varepsilon$. Deferring the adaption phase results in a decrease of skew, which means that the quality of inter-stream synchronization is increased. On the other hand, the deferred reaction increases the risk of buffer overflow or underflow, which may affect the quality of intra-stream synchronization. Consequently, the δ parameter, whose value may range from zero to the maximum delay of control messages, can be used to put emphasis on either inter-stream or intra-stream synchronization quality. We assume, however, that for a majority of applications δ may be set to zero, even in wide area networks.

3.5 Master switching protocol

In our protocol, we distinguish between two types of master switching. The first type of switching, called *policy-initiated*, is performed whenever (a change in) the synchronization policy requires a new assignment of the master role. In this case, the controller, which enforces the policy, performs the switching just by sending a *GrantMaster message* to the new master and a *QuitMaster message* to the old master. *GrantMaster* specifies the target buffer area of the new master, which is determined by the controller, depending on the chosen policy. With this simple protocol it may happen that for a short period of time there exist two masters, which both propagate *Adapt* messages. Our protocol prevents inconsistencies by performing *Adapt* requests in timestamp order (see below).

The second type of switching is *recovery-initiated*. A sink slave initiates recovery when its stream becomes critical. A stream is called critical if its current buffer delay is in a critical region and (locally) no rate adaption improving the situation is in progress. A very attractive property of our protocol is that a slave can immediately react when its stream becomes critical. Recovery goes as follows. First, the slave makes a transition to a so-called *tentative master* (or *t-master* for short) and informs the controller about this by sending an *IamT-Master* message. Then – without waiting for any response – it enters an adaption phase, in which it adapts release rate R_2 in a way that its buffer delay can be expected to move out of the critical region. In order to keep the other streams in sync, it propagates an *Adapt* request to all other sink agents, including the master. At the end of the adaption phase, a t-master falls back into the slave role. Should the stream still be critical by this time, then the recovery procedure is initiated once again.

Obviously, our protocol allows multiple instances to propagate *Adapt* concurrently, which may cause inconsistencies leading to the loss of synchronization if no care is taken. As already pointed out above, policy-initiated switch-

ing may cause the new master to send *Adapt* messages while the old master is still in place. Moreover, at the same point in time, there may exist any number of t-masters propagating *Adapt* requests concurrently. It should be clear that stream synchronization can be ensured only if *Adapt* messages are performed in the same order at each agent. This requirement can be fulfilled by including a timestamp in *Adapt* requests and performing these requests in timestamp order at the agent sites. The latter means that an agent accepts an *Adapt* request only if it is younger than all other requests received before. Older requests are just discarded.

However, performing requests in some timestamp order is not sufficient. Assume, for example, that the master and some t-master propagate *Adapt* requests at approximately the same time, and the former requests an increase of the release rate, while the latter requests a decrease. For some synchronization policies, this might be a very common situation (see for example the minimum delay policy described in the next section). If the timestamps were solely based on system time and the master would perform the propagation slightly after the t-master, then the t-master's request would be wiped out, although it is the reaction to a critical situation and hence is more important. The stability of the algorithm can only be guaranteed if recovery actions are performed with the highest priority.[1] Consequently, the timestamping scheme defining the execution order of *Adapt* requests must take into account the 'importance' of requests.

The precedence of *Adapt* requests sent at approximately the same time is given by the following list in increasing order: (1) requests of old masters, (2) requests of the new master (3) requests of t-masters. We apply a structured timestamping scheme to reflect this precedence of requests. In this scheme, a timestamp has the following structure: $< E_R \cdot E_M \cdot T >$, where E_R denotes a *recovery epoch*, E_M designates a *master epoch*, and T is the *real-time* when the message tagged with this timestamp was sent. A new recovery epoch is entered when a slave performs recovery, while a new master epoch is entered whenever a new master is selected. So, a recovery epoch may have seen several master epochs. As will be seen below, entering a new recovery epoch requires a new master to be selected.

Each control message contains a structured timestamp, which is generated before the message is sent on the basis of two local epoch counters and the local (synchronized) clock. The controller and the agents keep track of the current recovery and master epoch by locally maintaining two epoch counters. Whenever they accept a message whose timestamp contains an epoch value greater than the one recorded locally, the corresponding counter is set to the received epoch value. Moreover, an agent increments its local recovery epoch counter when it performs recovery, i.e., the *IamT-Master* message sent to the controller already reflects the new recovery period. The controller increments its master epoch counter when it selects a new master, i.e., the *GrantMaster* message already indicates the new master epoch.

[1] We assume that at no point in time there exist two t-masters that try to adapt the release rate in a contradicting fashion, i.e., one tries to increase the rate, while the other tries to decrease it. This is achieved by enabling mas ter switching only for one type of critical situation, underflow or overflow. Which type is enabled depends on the chosen sync policy (see Sect. 4)

Adapt requests are accepted only in strict timestamp order. Should an agent receive two requests with the same timestamps, total ordering is achieved by ordering these two request according to the requestors' unique identifiers included in the messages. As a slave performing recovery enters a new recovery epoch, all *Adapt* request generated by some master in the previous recovery epoch are wiped out. Similarly, selecting a new master enters a new master epoch, and by this wipes out all *Adapt* request from former masters. When a master receives an *Adapt* request indicating a younger master or recovery epoch, it can learn from this message that there exists a new master or a t-master performing recovery, respectively. In both cases, it immediately gives up the master role and becomes a slave.

As mentioned above, a critical slave sends an *IamT-Master* message when it becomes a t-master. When the controller receives such a message indicating a new recovery epoch, it must select a new master. Which stream becomes the new master primarily depends on the synchronization policy chosen. For example, the originator of the *IamT-Master* message establishing a new recovery epoch may be granted the master role. All other messages of this type belonging to the same recovery epoch are discarded upon arrival (see Fig. 6).

In summary, in an adaption phase a t-master or master may receive an *Adapt* or *GrantMaster* message. They are only accepted if they are younger than all other control messages of the same type received before. If an *Adapt* request is accepted, a new adaption phase is started based on the target media time included in the accepted request. As mentioned above, a master accepting an *Adapt* message immediately becomes a slave. If *GrantMaster* is accepted, the recipient becomes master and acts accordingly. A t-master that has not received *GrantMaster* by the end of the adaption phase goes back to the slave role. Of course, if it is still critical by this time, it initiates recovery again.

In the previous section, we discussed skew in the adaption phase without considering master switching. The possibility of switching the master role can increase the skew, as it may happen that the master and a t-master independently from each other decide to adapt in opposite directions. The worst case skew among sinks can be observed if such a decision is made at approximately the same time. The maximum skew can be shown to be

$$Skew_{max} = \max(0, d_m - \delta) \cdot (|R_{corr,master}| + |R_{corr,t-master}|) + \varepsilon,$$

where d_m denotes the transmission delay of *Adapt* messages and δ is the time the adaption phase is deferred. If δ is set to the maximum delay of control messages the skew is bounded by ε. The skew bound is increased by $d_m \cdot (|R_{corr,master}| + |R_{corr,t-master}|)$ if δ is zero. This term will be in the range of 20–30 ms in wide area networks and correspondingly lower in local area networks. Remember that if δ equals zero, streams may immediately perform adaptions at the time they become critical.

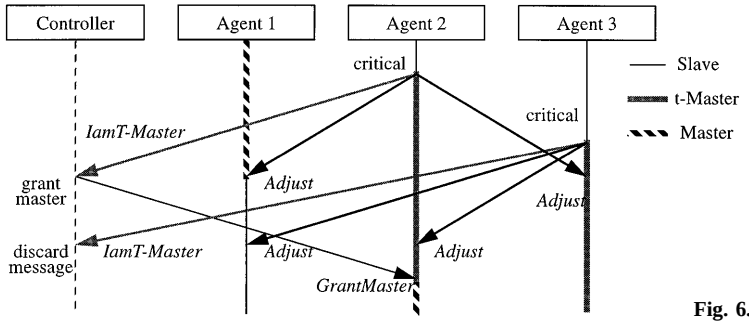


Fig. 6. Recovery-initiated master switching

4 Synchronization policies

ASP has many parameters for tuning the protocol to the characteristics of the underlying system as well as to the quality of service requested by the given application. A discussion of all these parameters would go far beyond the scope of this paper. Therefore, we will focus on the most important parameters, in particular those influencing the synchronization policy: the low- and high-water mark, the width of the target area and its placement in the play-out buffer, as well as the rules for granting the master role.

The intra-stream synchronization quality in terms of data loss due to underflow or overflow is primarily influenced by the *LWM* and *HWM* values. As pointed out in Sect. 2, the play-out time of a data unit u is $t_0 + \Delta + \mathrm{TS}(u)$, where Δ is adapted as needed in adaption phases. For a data unit released on time, the sum of its transfer delay and buffer delay must be equal to Δ. Assume, for example that the transfer delay of u is $d_T = \Delta - LWM$, i.e., u's buffer delay is at the border of the lower critical region. Obviously, if the transfer delays of the data units following u do not differ from d_T by more than *LWM*, there is no buffer underflow. Remember that Δ is immediately adapted when the buffer delay enters a critical region. Our experiments with ASP have shown that a reasonable value for the width of a critical region is $j/2$, where j denotes the jitter of the corresponding data stream.

Increasing *LWM* generally increases the intra-stream synchronization quality as the data loss probability is decreased. At the same time, however, this modification may increase the end-to-end delay of the sync group, which might be critical for certain applications. ASP allows the client to modify *LWM* and *HWM* values even while the presentation is in progress. For example, it is conceivable that a user interactively adjusts the stream quality during play-out. Alternatively, an internal mechanism similar to the one described in Kaeppner et al. (1994) may monitor the data loss rate and adjust the water marks as needed.

The width of the target area determines the aggressiveness of the buffer control algorithm. The minimum width of the target area is $\omega = c \cdot j$, where c depends on the smoothing function used to determine $d_B(t)$. In our experiments c turned out to be about 0.3. The larger the width of the target area, the less adaptions of the release rate are required. On the other hand, with a large target area, there is only limited control over the actual buffer delay. If, for example, the actual buffer delay has to be kept as close as possible to *LWM* to minimize the end-to-end delay, a small target area is preferable.

The location of the target area together with the way how the master role is granted are the major policy parameters of ASP. This will be illustrated by the following two policies, the minimum delay policy and the minimum loss policy.

The goal of the *minimum delay policy* is to achieve the minimum end-to-end delay for a given intra-stream synchronization quality. To reach this goal, the stream with the currently maximum transfer delay is granted the master role, and this stream's buffer delay is kept as close as possible to *LWM*. This means that the target area for the master is located as follows: $LTB = LWM$ and $UTB = LWM + \omega$, where ω is the jitter of the smoothed buffer delay $d_B(t)$.

Due to changing network conditions, it may happen that the transfer delay of a slave stream surpasses the one of the master. This will cause the slave's buffer delay to fall below its *LWM*, triggering recovery. When the controller receives an *IamT-Master* message, it assigns the master role to the received message's originator by sending a *GrantMaster* request. If it receives multiple *IamT-Master* messages originated in the same recovery epoch, only the first one is accepted, all the other ones are ignored. This strategy ensures that the stream with the maximum transfer delay always becomes master. The end-to-end delay of the sync group at time t amounts to the maximum transfer delay at t plus $(UTB + LTB)/2$, which is the minimum end-to-end delay that can be achieved at t.

With the minimum delay policy, a slave running out of buffer may cause master switching to be performed continuously. To ensure stability in those situations, master switching is disabled for overflow-critical streams. Various policies for a slave to recover from overflow-critical situations are possible (for details, see Sect. 5).

The possibility of adjusting *LWM* dynamically makes this policy very powerful. By increasing *LWM*, the data loss rate is decreased, while the end-to-end delay is increased. The loss rate is increased and the end-to-end delay is decreased if *LWM* is decreased. Consequently, by dynamically adjusting *LWM*, the user may (interactively) determine the appropriate trade-off between end-to-end delay and intra-stream synchronization quality.

While the minimum delay policy minimizes the buffer delay, the *minimum loss policy* maximizes the buffer delay to minimize the probability of buffer underflow for the available buffer space. This policy is appropriate for those applications, for which a perfect transmission (i.e., low loss rate) is more important than a low end-to-end delay.

With this policy, the stream with the at present minimum transfer delay is granted the master role. The master's buffer

delay is kept as close as possible to *HWM*, which means that the target area for the master is located as follows: *UTB* = *HWM* and *LTB* = *HWM* $- \omega$, where ω denotes the jitter of $d_B(t)$. Note that each slave stream has a lower buffer delay than the master stream, as the latter is the one with the minimum transfer delay.

When changing network conditions cause a slave to experience a smaller transfer delay than the current master, this slave's buffer delay will exceed *HWM*, triggering recovery. The controller receiving an *IamT-Master* message reacts in exactly the same way as with the previous policy. It sends a *GrantMaster* message to the originator of the *IamT-Master* message arriving first in a recovery period, all following messages belonging to the same recovery period are ignored. Obviously, this policy ensures that always the stream with the minimum transfer delay is the master. Maximizing the buffer delay of the master means keeping the buffers as full as possible and thereby minimizing the loss probability due to underflow.

With the minimum loss policy a "starving" slave stream may cause master switching to be performed continuously. To ensure stability in those situations, master switching is disabled for underflow-critical streams if this policy is applied. Stability aspects and recovery for critical streams are discussed in detail in the next section.

5 Stability and buffer requirements

ASP uses buffering to equalize the different transfer delays of the streams in a sync group. Therefore, the size of the play-out buffer of an individual stream depends on the delay characteristics of the stream group.

The streams in a sync group may have different buffer requirements. We will determine the size of the streams' play-out buffer in terms of time units to keep the results independent from the encodings of the various media. Let $d_{i,max}$ and $d_{i,min}$ be the maximum and minimum transfer delay of stream i, respectively, and $\delta_{i,k} = d_{k,max} - d_{i,min}$. The target of master stream k is $LWM_k + \omega_k/2$, where ω_k is the width of k's target area. Stream i's high water mark can be determined as follows: $HWM_i = max(LWM_k + \omega_k/2 + \delta_{i,k} : k \in G - \{i\})$, where G denotes the corresponding sync group. Consequently, the size of the play-out buffer of stream i is $B_i = HWM_i + LWM_i$ assuming the same width for both critical regions.

The buffer size is determined based on assumptions concerning the maximum and minimum transfer delay. If the underlying network provides (reasonable) delay guarantees and buffer is allocated according to the results above, it may never happen that two streams of a sync group are critical in a contradicting way, i.e., one experiences a buffer underflow, while the other suffers from overflow at the same time. If, however, the underlying network does not provide a deterministic service, the assumed minimum and maximum delays have to be determined on a statistical basis. In this case, it might happen that a sync group's streams experience underflow and overflow at the same time. We will call this an underflow&overflow situation.

It is important to note that an underflow&overflow situation does not jeopardize the stability of ASP. Since the minimal delay and minimal loss policy both enable master switching either for underflow recovery or for overflow recovery, an underflow&overflow situation may never cause master switching to be performed continuously. For example, consider the minimum delay policy. Remember, this policy minimizes the buffer delays of all streams in a sync group by minimizing the buffer delay of the stream with the currently longest transfer delay. For this policy master switching is only enabled for buffer underflow. While a stream experiencing an underflow will always initiate master switching and decrease the stream's play-out rate accordingly, the recovery processing for overflow depends on the policy implemented by the stream's sink agent. Following policies are conceivable:

Dynamic buffer allocation. In order to avoid overflow, the buffer is dynamically extended when a stream's buffer delay exceeds *HWM*. The dynamically allocated buffer can be released as the buffer delay decreases due to changing network conditions. If dynamic buffer allocation is impossible there are two remaining policies, skipping and stream removal.

Skipping. The sink agent may skip data units, either already residing in the buffer or just arriving. Of course, if data units differ in importance (e.g., I-, B- and P-frames of MPEG videos), the agent will try to skip the less important ones first. Obviously, this policy causes data loss, and hence decreases the quality of the individual stream, while the quality of inter-stream synchronization is not affected.

Stream removal. When a stream becomes (overflow-) critical, the stream's sink agent may remove the stream temporarily from the sync group. This removal is a local operation that does not require any communication with other protocol instances. After removal the agent can adjust the play-out rate independent from the other streams in the sync group. However, it still receives the *Adapt* requests from the master and thus is able to keep track of the sync group's media time. Stream removal will cause the stream's (local) media time to differ from the sync group's media time. In other words, this policy decreases the quality of inter-stream synchronization, while the quality of the individual streams is not affected. The skew can be minimized by keeping the buffer delay of the removed stream close to *HWM*. A removed stream may rejoin the sync group when its local media time equals the sync group's media time.

Obviously, skipping and stream removal can be combined. For example, an agent may perform skipping until the loss rate reaches a certain threshold and then switch to stream removal.

In our discussion above, we have confined ourself to the minimum delay policy, as the stability arguments for the minimum loss policy are almost symmetrical.

6 Simulation results and performance measurements

In order to investigate ASP's behavior in different environments, the proposed protocol has been simulated extensively. Moreover, it has been implemented and its performance has been experimentally measured (for details, see Helbig 1996). In this section, we will discuss the major results of this work,

focusing on ASP's ability to adapt to changing conditions, its message overhead and skew.

Our simulations use delay data measured in the Internet as well as synthetically generated delays. The Internet data are used to investigate ASP's behavior in fairly unpredictable environments, while the synthetic data allow for more systematic investigations.

In our first simulation, the transfer delays are based on measurements in the Internet. This simulation illustrates how ASP reacts on a client-initiated reduction of the end-to-end delay (Fig. 7a–d). The target area in the play-out buffer is defined by *LTB* = 100 ms and *UTB* = 200 ms. This setting leads to a constant release rate and an end-to-end delay of about 260 ms. There is no data loss due to late arrivals. During the simulation, the target area is moved to *LTB* = 35 ms and *UTB* = 135 ms to reduce the end-to-end delay by about 90 ms. This reduction causes an increase in late arrivals by approximately 2.5%. This client-initiated adaption is achieved within a single adaption phase.

The following simulations use synthetic transfer delays generated according to a normal distribution.[2] The transfer delay distributions of streams S1, S2 and S3 have a mean transfer delay/ standard deviation of 200 ms/20 ms, 180 ms/10 ms, 200 ms/10 ms, respectively. We have chosen similar transfer delays, as this is the interesting case with regard to the frequency of master switching.

The simulation results depicted in Fig. 8a show the dependency of the end-to-end delay and the data loss due to late arrivals. Parameter *LWM* is set to 10, 20, 50, 100, and 200 ms, respectively. If *LWM* is increased, this increases the end-to-end delay and reduces the number of late data units, e.g., for stream S3 from 10% to 0%. Our simulations show that increasing *LWM* beyond 50 ms does not improve the quality of the considered streams anymore.

During adaption phases, the skew is determined by the size of rate corrections and the transfer delay of *Adapt* messages. Figure 8b illustrates the impact of the length of the adaption phase on the minimum, average and maximum rate correction R_{corr}. The results show that a reasonable length of the adaption phase is from 1 to 5 s, leading to a maximum rate correction of about 2% and an average rate correction below 1%. The maximum rate correction for a length of 5 s is about 0.35%.

The resulting skew during adaption phases is clearly below the values tolerated in the scenarios described in the experiments of Steinmetz and Engler (1993). With the available clock synchronization protocols, such as NTP (Mills 1990), we can assume clocks to be synchronized within the lower milliseconds range. By using radio-controlled clocks, this situation will improve even further. The skew added by ASP for rate corrections of up to 2% is typically below 1 ms in a LAN and below 20 ms in a WAN, assuming transfer delays of up to 1 s.

Finally, we will investigate how ASP adapts to changing transfer delays. We will consider two types of changes, a jump and a ramp-shaped change. For the jump, the height is varied in steps of 10 ms from -50 ms to +50 ms, while for the ramp, the transfer delay is continuously increased within a certain time interval. The length of the time interval is varied from 1 s to 50 s, and ramp heights of 10, 20 and 50 ms are considered. In all simulations, the width of the target area is 20 ms and the adaption phase is 5 s in length.

Figure 9a shows the results of the jump simulation. Jumps up to half the width of the target area either cause no or a single rate adaption, depending on the buffer delay at the time of the jump. Consequently, 0.5 rate adaptions are required in average. Jump heights of 20 ms (width of the target area) and 50 ms require two and three adaption phases, respectively. The reason why multiple adaption phases are needed is the smoothing function applied on buffer delays, which causes the first rate adaption to be smaller than actually needed.

In Fig. 9b, the simulation results for the ramp-shaped delay changes are illustrated. Independent of the length of the interval, changes of half the target area width lead to a single or no rate adaption, and a change of the same size as the target area requires two adaption phases. Only larger changes over longer time intervals require more rate adaptions since they cause a sequence of small adaptions. For the 50-ms ramp, the worst case is 8 adaptions in 50 s.

To verify the simulation results, ASP has been implemented and evaluated in the CINEMA project (Configurable Integrated Multimedia Architecture; Rothermel et al. 1994). CINEMA provides a platform for developing and controlling multimedia applications in distributed environments. In particular, it offers abstractions and mechanisms to build distributed multimedia applications by configuration of basic processing and communication elements. Synchronization constraints between streams may be specified by means of so-called clock hierarchies (Rothermel and Helbig 1996). While clock hierarchies are programming abstractions, ASP is the mechanism that actually performs stream synchronization. CINEMA runs on IBM RS/6000 workstations under AIX, as well as Sun SPARCstations under Solaris.

So far, measurements have been performed for two network technologies, a 10-Mbps Ethernet and a 155-Mbps ATM network. With these measurements we could confirm the essential results of our simulations (for details, see Helbig 1996). In the Ethernet-based experiments, *Adapt* messages are generated every 10–20 s for rather tight target areas. By increasing the target area, it can be achieved that *Adapt* messages are sent only every couple of minutes. The maximum rate correction R_{corr} is below 2%, average rate corrections are between 0.4% and 1.2%. Consequently, the skew added by ASP is far below the skew limits given in Steinmetz and Engler (1993) for scenarios such as lip synchronization or video/text overlays. As expected, experiments performed in the ATM environment show even better results. Measurements in WAN environments are subject to future work.

7 Related work and conclusions

Existing approaches to stream synchronization can be classified in various ways. One classification criterion is whether or not synchronization is distributed. In the case of distributed approaches, the sinks of the sync group may reside

[2] Normal distribution for packet delays in packet-switched networks is suggested in Alvarez-Cuevas et al. (1993) and Shivakumar et al. (1995)

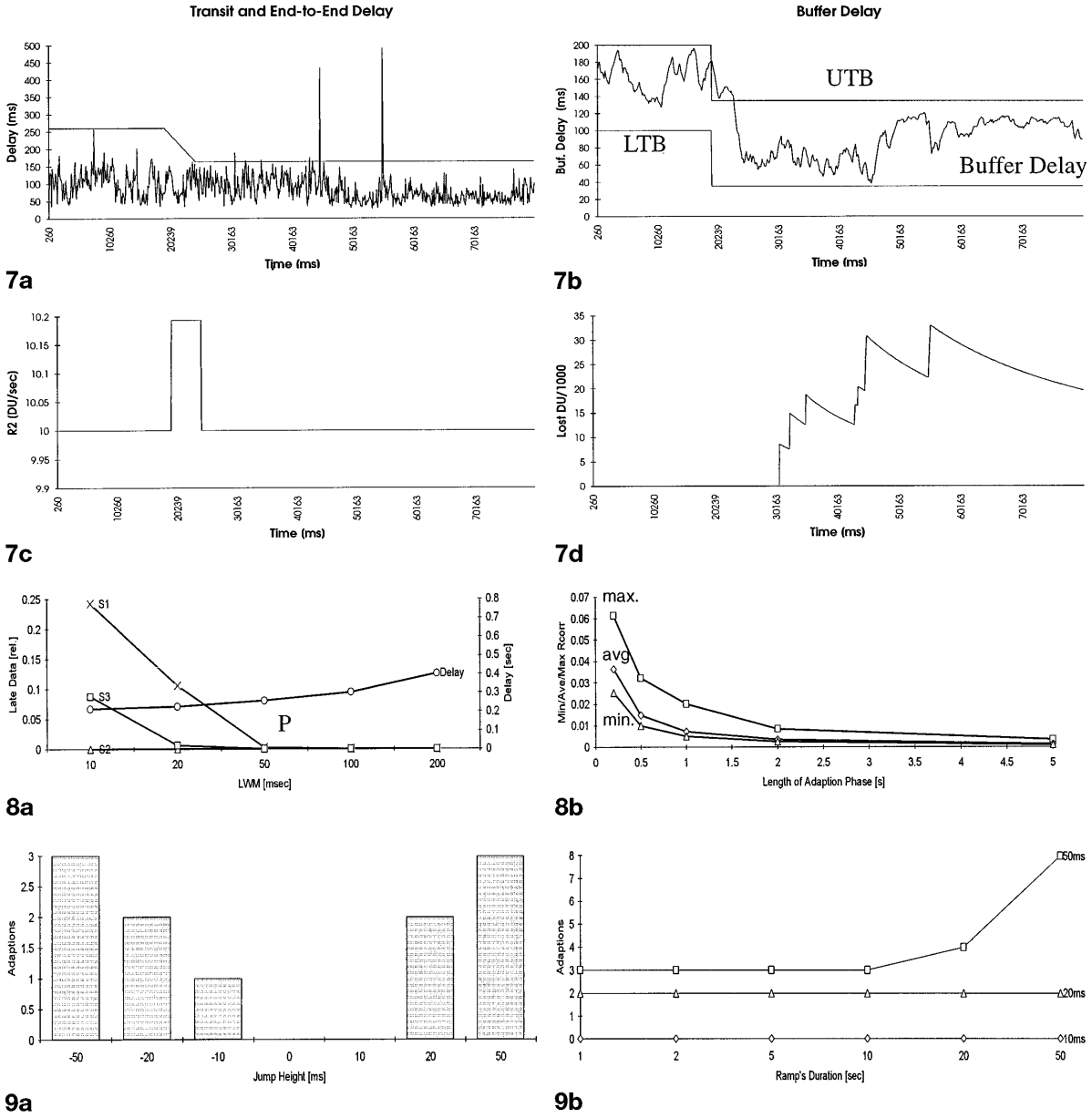


Fig. 7. a Transmission and end-to-end delay. **b** Buffer delay of master stream. **c** Release rate of master stream. **d** Late data units/1000

Fig. 8. a Delay versus late data units. **b** R_{corr} versus length of adaption phases

Fig. 9. a Reaction on jump in delay. **b** Reaction on ramp in delay

on different nodes, while local approaches require all sinks to reside on the same node.

The class of local approaches comprises a number multimedia toolkits, e.g., ACME (Anderson and Homsy 1991), Multimedia Presentation Manager (IBM 1992), QuickTime (Apple 1991), or Tactus (Dannenberg et al. 1992), as well as various synchronization algorithms proposed in the literature; e.g., Ravindran and Bansal (1993), Kaeppner et al. 1994), (Shivakumar et al. (1995). Distributed approaches include algorithms proposed in Ramanathan and Rangan (1992), Agarwal and Son (1994), the Flow Synchronization Protocol (Escobar et al. 1994), the Lancaster Orchestration Service (Campell et al. 1992), as well as ASP.

Both local as well as distributed approaches may be rigid or adaptive. For example, the Concord algorithm (Shivakumar et al. 1995) and the DMOS protocol (Kaeppner et al. 1994) fall into the class of local adaptive approaches. The Concord algorithm allows to trade off packet loss rates, end-to-end delay and skew. The algorithm computes the packet delay distribution on-the-fly and delivers it to the client which decides on adaptions. In other words, the algorithm itself does not provide for automatic adaptions. In DMOS, a QoS parameter “rate of late data units” is monitored, allowing applications to trade off end-to-end delay versus loss rate. Automatic adaptions are performed as required. In both schemes, inter-stream synchronization is based on comput-

ing a reference end-to-end delay for all streams by a dedicated (centralized) entity. Transfering this approach to distributed settings would lead to a significant message overhead for collecting state information and propagating control messages.

Distributed adaptive approaches may be based on local time or global time, where the latter is achieved by clock synchronization. No global time is required for the algorithms proposed in Ramanathan and Rangan (1992) and Agarwal and Son (1994). Stored data streams are transferred from a centralized server to distributed sinks. The sinks are required to periodically send feedback messages to the server, which uses these messages to estimate the temporal state of the individual streams. In Ramanathan and Rangan (1992), the accuracy of these estimations depends on the jitter of feedback messages. Agarwal and Son (1994) eliminate this dependency by estimating the differences between system clocks by means of probe messages. With this modification, accuracy depends on the jitter of probe messages. The feedback messages cause an overhead of n messages per period for n streams. After a stream becomes critical, it takes at least one message round-trip time before an adaption takes effect at the sink.

Both the Flow Synchronization Protocol (Escobar et al. 1994) and the Lancaster Orchestration Service (Campell et al. 1992) are distributed adaptive approaches assuming synchronized clocks. In the Flow Synchronization Protocol, each sink periodically sends its delay estimate to all other sinks in the sync group. Having received all delay estimates, each sink locally performs the same function on its own and the received estimates to determine the end-to-end delay for the next period. The message complexity is $n \cdot (n-1)$ messages (or n messages if multicast is available) per period, however, various optimizations are proposed to reduce this message overhead. When a stream becomes critical, its sink cannot perform (global) adaptions before the next period begins.

With Lancaster Orchestration Service, a centralized controller periodically receives the temporal state of each sink in the sync group. Based on the collected information, the controller periodically decides whether adaptions are needed and sends the corresponding adapt requests. The message overhead per period is at least n messages, and $2 \cdot n$ in the worst case. Moreover, reactions on critical situations are deferred by at least one message round-trip time.

ASP belongs to the same class as the Flow Synchronization Protocol and the Lancaster Orchestration Service. The major difference is that ASP does not know the concept of a period. Instead of sending control messages periodically, in ASP adapt requests are sent solely on demand, when rate adaptions actually become necessary due to changing network conditions or QoS requirements. The propagation of adapt requests requires $(n-1)$ messages (or 1 message if multicast is available). A nice feature of ASP is that a sink may react immediately on critical situations. The price of this feature is an increase in skew, which, however, can be ignored for most applications as has been shown in the previous section. For applications that are extremely skew-sensitive, ASP provides the possibility to defer adaptions artificially in order to avoid this skew.

ASP is a very general and flexible synchronization mechanism that can be tailored to various network characteristics, as well as to a wide range of multimedia applications. ASP has been simulated and implemented in the CINEMA system. Both the simulations and the performance measurements confirmed the properties postulated for ASP.

References

Agarwal N, Son S (1994) Synchronization of distributed multimedia data in an application-specific manner. In: 2nd ACM International Conference on Multimedia, San Francisco, Calif., pp 141–148

Alvarez-Cuevas F, Bertram M, Oller F, Selga JM (1993) Voice synchronization in packet switching networks. IEEE Network 7: 20–25

Anderson DP, Homsy G (1991) Synchronization policies and mechanisms in a continuous media i/o server. Report No. UCB/CSD 91/617, Computer Science Division (EECS), University of California, Berkeley, Calif.

Apple (1991) QuickTime Developer's Guide. Apple Computer Inc., Cupertino, Calif., USA

Campell A, Coulson G, Garcia F, Hutchison D (1992) A continuous media transport and orchestration service. SIGCOMM'92 Communications Architectures and Protocols. pp 99–110

Clark DD, Shenker S, Zhang L (1992) Supporting real-time applications in an integrated services packet network: Architecture and mechanism. SIGCOMM'92 Communications Architectures and Protocols. pp 14–26

Dannenberg RB, Neuendorffer T, Newcomer JM, Rubine D (1992) Tactus: Toolkit-level support for synchronized interactive multimedia. 3rd International Work-shop on Network and Operating System Support for Digital Audio and Video. pp 264–275

Escobar J, Partridge C, Deutsch D (1994) Flow synchronization protocol. IEEE Trans Networking 2: 111–121

Ferrari D (1990a) Client requirements for real-time communication services. Request for Comments RFC 1193

Ferrari D (1990b) Design and applications of a delay jitter control scheme for packet-switching internetworks. In: 2nd International Workshop on System Support for Digital Audio and Video, Heidelberg, Germany.

Helbig T (1996) Communication and synchronization of multimedia data streams in distributed systems (in German). PhD thesis, University of Stuttgart, Faculty of Computer Science, Stuttgart, Germany

IBM (1992) Multimedia Presentation Manager Programming Reference and Programming Guide 1.0, IBM Form: S41G-2919-00 and S41G-2920-00. IBM Corporation

Käppner T, Henkel F, Müller M, Schröer A (1994) Synchronisation in einer verteilten Entwicklungs- und Laufzeitumgebung für multimediale Anwendungen. Innovationen bei Rechen- und Kommunikationssystemen. pp 157–164

Köhler D, Müller H (1994) Multimedia playout synchronization using buffer level control. 2nd International Workshop on Advanced Teleservices and High-Speed Communication Architectures, Heidelberg, Germany.

Mills DL (1990) On the accuracy and stability of clocks synchronized by the network time protocol in the internet system. Comput Commun Rev 20: 65–75

Postel (1981) Transmission control protocol, darpa internet program, protocol specification. RFC 793

Ramanathan S, Rangan PV (1992) Continuous media synchronization in distributed multimedia systems. 3rd International Workshop on Network and Operating System Support for Digital Audio and Video. pp 289–296

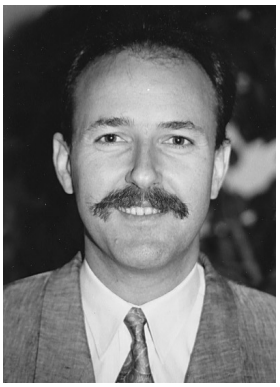
Ravindran K, Bansal V (1993) Delay compensation protocols for synchronization of multimedia data streams. IEEE Trans Knowl Data Eng 5: 574–589

Rothermel K, Barth I, Helbig T (1994) CINEMA – an architecture for distributed multimedia applications. In: Architecture and Protocols for High-Speed Networks. Kluwer, Dordrecht, pp 253–271

Rothermel K, Helbig T (1996) Clock hierarchies: An abstraction for grouping and controlling media streams. IEEE J Select Areas Commun (Synchronization Issues in Multimedia Communications) 14: 174–184

Shivakumar N, Sreenan C, Narendran B, Agarwal P (1995) The concord algorithm for synchronization of networked multimedia streams. In: IEEE International Conference on Multimedia Computing and Systems, Washington, D.C. pp 31–40

Steinmetz R, Engler C (1993) Human perception of media synchronization. Technical Report 43.9310, IBM ENC, Heidelberg, Germany



KURT ROTHERMEL received his doctoral degree in Computer Science from Stuttgart University in 1985. From 1986 to 1987 he spent a sabbatical at the IBM Almaden Research Center, working on distributed database management systems. In 1988 he joined IBM‘s European Networking Center, where he was responsible for several projects in the area of distributed application systems. He left IBM in 1990 to become a Professor for Computer Science back at Stuttgart University, where he now leads the Distributed Systems Research Group. His current research interests are communication architectures and protocols, distributed multimedia systems, management of distributed systems, and mobile software agents. He is a member of IEEE Computer Society, ACM and GI.

TOBIAS HELBIG studied Computer Science at the University of Stuttgart. He received his M.Sc. (Diplom-Informatiker) degree in 1992. At the same university he studied towards his PhD in Computer Science in the years 1993–96. He is now a research scientist with the Philips Research Laboratories in Aachen, Germany. His main research interests are multimedia system services, control and synchronization of continuous data streams in distributed environments and QoS handling.