An Adaptive Stream Synchronization Protocol

Kurt Rothermel, Tobias Helbig

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

Abstract. Protocols for synchronizing data streams should be highly adaptive with regard to both changing network conditions as well as to individual user needs. The Adaptive Synchronization Protocol we are going to describe in this paper supports any type of distribution of the stream group to be synchronized. It incorporates buffer level control mechanisms allowing an immediate reaction on overflow or underflow situations. Moreover, the proposed mechanism is flexible enough to support a variety of synchronization policies and allows to switch them dynamically during presentation. Since control messages are only exchanged when the network conditions actually change, the message overhead of the protocol is very low.

1 Introduction

In multimedia systems, synchronization plays an important role at several levels of abstraction. At the data stream level, synchronization relationships are defined among temporally related streams, such as a lip-sync relationship between an audio and a video stream. To ensure the synchronous play-out of temporally related streams, appropriate stream synchronization protocols are required.

Solutions to the problem of data stream synchronization seem to be quite obvious, especially if clocks are synchronized. Nevertheless, designing an efficient synchronization protocol that is highly adaptive with regard to both changing network conditions and changing user needs is a challenging task. If the network cannot guarantee bounds on delay and jitter, or a low end-to-end delay is of importance, the protocol should operate on the basis of the current network conditions rather than some worst case assumptions, and should be able to automatically adapt itself to changing conditions. Moreover, the protocol should be flexible enough to support various synchronization policies, such as 'minimal end-to-end delay' or 'best quality'. This kind of flexibility is important as different applications may have totally different needs in terms of quality of service. In a teleconferencing system, for example, a low end-to-end delay is of paramount importance, while a degraded video quality may be tolerated. In contrast, in a surveillance application, one might accept a higher delay rather than a poor video quality.

Protocols for synchronizing data streams can be classified into those assuming the existence of synchronized clocks and those making no such assumption. The Adaptive Synchronization Protocol (ASP), we are going to propose in this paper, belongs to the first class and has the following characteristics:

- ASP supports any kind of distribution of the group of streams to be synchronized, i.e. the sources of the streams as well as their sinks may reside on different nodes. Streams may be point-to-point or point-to-multipoint.
- ASP incorporates buffer level control mechanisms and by this is able to react immediately on changing network conditions. It allows a stream's play-out rate to be adapted immediately when the stream becomes critical, i.e. when it runs the risk of a

buffer underflow or overflow. If changing network conditions cause several streams to become critical at the same time, each stream may immediately initiate the required adaption, independent from all other streams. Note that this property may improve the intrastream synchronization quality substantially.

- ASP monitors the network conditions indirectly by means of a local buffer level control mechanism and performs rate adaptions only if they are actually required, i.e. only when a stream becomes critical. Due to this fact, the overhead for exchanging control messages is almost zero if the streams' average network delay and jitter are rather stable.
- ASP supports the notion of a master stream, where the master controls the advance of the other streams, called slaves. The role of the master is assigned in accordance with the chosen synchronization policy and can be changed dynamically during the presentation if needed.
- ASP is a powerful and flexible mechanism that forms the base for various synchronization policies. It is powerful in the sense that the realization of a desired policy is a simple task: A policy is determined by setting a set of parameters and assigning the master role appropriately. For a chosen policy ASP can be tuned individually to achieve the desired trade-off between end-to-end delay and intrastream synchronization quality. This tuning and even the applied policy can be changed dynamically during the presentation.

The remainder of this paper is structured as follows. After a discussion of related work in the next section, the basic assumptions and concepts of ASP are introduced in Sect. 3. Then, Sect. 4 presents ASP by describing its protocol elements for start-up, buffer control, master/slave synchronization and master switching. We show in Sect. 5 how different synchronization policies can be efficiently realized on top of the proposed synchronization mechanism, and provide some simulation results illustrating the performance of ASP in Sect. 6. Finally, we conclude with a brief summary.

2 Related Work

The approaches to stream synchronization proposed in literature differ in the stream configurations supported. Some of the proposals require all sinks of the synchronization group to reside on the same node (e.g., Multimedia Presentation Manager [5], ACME system [2]). Others assume the existence of a centralized server, which stores and distributes data streams. The scheme proposed by Rangan et al. [10], [11] plays back stored data streams from a server. Sinks are required to periodically send feedback messages to the server, which uses these messages to estimated the temporal state of the individual streams. Since clocks are not assumed to be synchronized, the quality of these estimations depends on the jitter of feed-back messages, which is assumed to be bounded. A similar approach has been described in [1], which requires no bounded jitter but estimates the difference between clocks by means of probe messages.

Both the Flow Synchronization Protocol [4] and the Lancaster Orchestration Service [3] assume synchronized clocks and support configurations with distributed sinks and sources. However, neither of the two protocols allows a sink to react immediately when its stream becomes critical. Moreover, the former protocol does not support the notion of a master stream, which excludes a number of synchronization policies. Finally, both schemes do not provide buffer level control concepts at their service interfaces, which makes the specification of policies more complicated than for ASP.

Some buffer level control schemes have been proposed also. The scheme described in [7] aims at intrastream synchronization only. In [6], stream quality is defined in terms of the rate of data loss due to buffer underflow. A local mechanism is proposed that allows either to minimize the stream's end-to-end delay or to optimize its quality.

3 Basic Concepts and Assumptions

The set of streams, which are to be played out in a synchronized fashion is called *synchronization group* (or sync group for short). ASP distinguishes between two kinds of streams, the so-called *master* and *slave streams*. Each sync group comprises a single master stream and one or more slave streams. While the rate of the master stream can be individually controlled, the ones of the slave streams are adapted according to the progress of the master stream. The master and slave role can be switched dynamically as needed.

For each sync group there exists a single synchronization server and several *clients*, two for each stream. The server 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 the switching of the master role. Moreover, it is this entity that enforces the synchronization policy chosen by the user. The server communicates with the clients, which are software entities controlling individual streams. Each stream has a pair of clients, a sink client and a source client, which are able to start, stop, slow-down or speed-up the stream. Depending on the type of stream it is controlling, a sink client either acts as a *master* or *slave*. To achieve interstream synchronization, the master communicates with its slaves according to a synchronization protocol.

ASP supports arbitrarily distributed configurations: A sync group's sources may reside on different sites, and the same holds for the sinks. The location of the server may be chosen freely, e.g., it may be located on the node that hosts the most sink clients.

We will assume that control messages are communicated reliably and 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 NTP [8], achieve clock synchronization with ε in the lower milliseconds range.

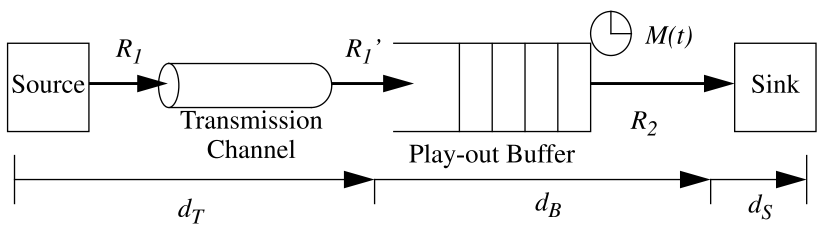

Fig. 1. Data Stream and Delay Model

The basic principle of interstream synchronization adopted by ASP and various other protocols based on the notion of global time (e.g., [4]) is very simple: Each data unit of a stream is associated with a timestamp, which defines its media time. To achieve synchronous presentations of streams, the streams' media time must be mapped to global time, such that data units with the same timestamp will be played out at the same (global) time. Similarly, the sources exploit the existence of synchronized clocks: data units with the same timestamp are sent at the same (global) time. Different transmission delays that may exist between different streams are equalized by buffering data units appropriately at the sink sites.

Our model of stream transmission and buffering is depicted in Fig. 1. 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 via an unidirectional transmission channel. The transmission channel introduces a certain delay and jitter, resulting in a *modified arrival rate* R_1'. At the sink's site, data units are buffered in a play-out buffer, from which they are released with a *release rate* R_2. The release rate, which determines how fast the stream's presentation advances, is directly controlled by ASP to manipulate the fill state of the play-out buffer and to ensure synchrony.

On its way from generation to play-out, a data unit is delayed at several stages. It takes a data unit a *transmission delay* d_T until it arrives in the buffer at the sink's site. This includes all the times for generation, packetization, network transmission and transfer into the buffer. In the buffer, a data unit is delayed by a *buffer delay* d_B before it is delivered to the sink device. In the sink, a data unit may experience a *play-out delay* d_S before it is actually presented.

The *media time* $M(t)$ specifies the stream's temporal state of play-out. It is derived from the timestamp *TS* of the data unit which is the next to be read out from the play-out buffer and the actual *play-out delay* d_S of the sink: $M(t) = TS - d_S$. However, the granularity of media time were too coarse would it simply be based on timestamps. Due to this fact, media time is interpolated between timestamps of data units to achieve a finer granularity

4 The Adaptive Synchronization Protocol

ASP can be separated into four rather independent subprotocols. After a brief overview, the start-up protocol, buffer control protocol, master/slave synchronization protocol, and master switching protocol are described in detail. It is important to point out, that this section concentrates on mechanisms, while possible policies exploiting these mechanisms will be discussed in the next section.

4.1 Overview

The *start-up protocol* initiates the processing of the sinks and sources in a given sync group. In particular, it ensures that the sources synchronously start the transmission and the sinks synchronously start the presentation. Start-up is coordinated by the server, 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 clients.

The *buffer control protocol* is a purely local mechanism, which keeps the fill state of the master stream's play-out buffer 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. Whenever the fill state moves out of the given target area, the buffer control protocol regulates the progress of the master stream by manipulating release rate R_2 accordingly.

The *master/slave synchronization protocol* ensures interstream synchronization by adjusting the progress of slave streams to the advance of the master stream. Processing of this protocol only involves a sync group's sink clients, one of them acting as master and the other ones acting as slaves. Whenever the master changes release rate R_2, it computes for some future point in time, say *t*, the master's media time $M(t)$, taking into account the modified value of R_2. Then, $M(t)$ and *t* are propagated in a control message to all slaves. When a slave receives such a control message, it locally adjusts R_2 in a way that its stream will reach *M(t)* at time *t*. Obviously, this protocol ensures that all streams are in sync again at time *t*, within the margins of the accuracy provided by clock synchronization. Notice that this protocol does not involve the server and is only initiated when the buffer situation or - in other words - the network conditions have changed.

The *master switching protocol* allows to switch the master role from one stream to another at any point in time. The protocol involves the server and the sink clients, whereas the server is the only instance that may grant the master role. Switching the master role may become necessary when the user changes its synchronization policy or some slave stream enters a critical state, i.e. runs the risk of having a buffer underflow or overflow. A nice property of this protocol is that a critical slave can react immediately by becoming a so-called tentative master, which is allowed to adjust R_2 accordingly. 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.

4.2 Start-up Protocol

Our start-up procedure is very similar to that described in [4]. The server initializes the synchronous start-up of a sync group's data streams by sending *Start* messages to each sink and source client. Each *Start* message contains besides other information a start-up time. All source clients receive the same start-up time, at which they are supposed to start transmitting data units. Similarly, all sink clients receives the same start-up time, which tells them when to start the play-out process.

Starting clients 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 message transmission delay d_m experienced by *Start* messages, and processing delays d_{proc} at the server site: $t_0 = t_{now} + d_m + d_{proc}$. Start-up of sinks is delayed by an additional time to allow the data units to arrive at the sinks' locations and to preload buffers. This delay, called expected delay d_{exp}, is computed from average delays $d_{ave,i}$ of the sync group's streams and the buffer delay $d_{pre,i}$ caused by preloading: $d_{exp} = max\ (d_{ave,i} + d_{pre,i})$, where $d_{pre,i}$ primarily depends on stream *i*'s jitter characteristic. We assume some infrastructure component that provides access to the needed jitter and delay parameters.

A *Start* message sent to a source client (at least) contains start time t_0 and the nominal rate R_1. *Start* received by a sink encompasses the start time $t_0 + d_{exp}$, the release rate $R_2 = R_1$ and a flag assigning the initial role (i.e. master or slave). Furthermore, it includes some initial parameters concerning the play-out buffer: the low water mark, high water mark and - in case of the master stream - the initial target area (see below).

Each client starts stream transmission or play-out at the received start-up time. Therefore, the start-up asynchrony 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.

4.3 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 current time *t*. The buffer delay at a given point in time is determined by the amount of buffered data. 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 [9]: $d_B(t_i) = \alpha\, d_B(t_{i-1}) + (1-\alpha)\, ActBufferDelay(t)$, or the Finite Impulse Response Filter as used in [7].

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 intrastream synchronization is primarily determined by the *LWM* and *HWM* values (for details see Sect. 5).

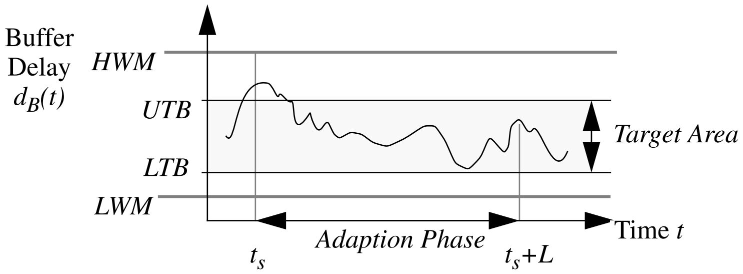

Fig. 2. Buffer Delay Adaption

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)*. 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. 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 server) 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, release rate R_2 is modified accordingly. The adapted release rate is $R_2 + R_{corr}$, where $R_{corr} = (d_B(t) - (LTB + (UTB-LTB)/2)) / L$. Length L of the adaption phase determines how aggressively the algorithm reacts. At the end of the adaption phase, it is checked whether or not $d_B(t)$ is within the target area. If it is in the target area, R_2 is set back to its previous value, the nominal rate R_1. Otherwise, the master immediately enters a new adaption phase.

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.

4.4 Master/Slave Synchronization Protocol

The master/slave synchronization protocol ensures that the slave streams are played out in sync with the 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 involves all sink clients, each of which acts either as master or slave.

Whenever it enters an adaption phase, the master 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 $M(t_s+L) = M(t_s) + L\cdot(R_2 + R_{corr})$. Subsequently, the master propagates an *Adapt* message to each slave in the sync group. This message includes the following information: end time $t_e=t_s+L$ of the adaption phase, target media time $M(t_e)$ at the end of the adaption phase, and a structured timestamp for ordering competing *Adapt* messages (see next section).

When a slave receives an *Adapt* message, it immediately enters the adaption phase by modifying its release rate R_2 according to the received target media time (see Fig. 3). The modified release rate is $R_2 = (M(t_e)\text{-}M(t_a)) / (t_e - t_a)$, 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 its previous value, the nominal stream rate. 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.

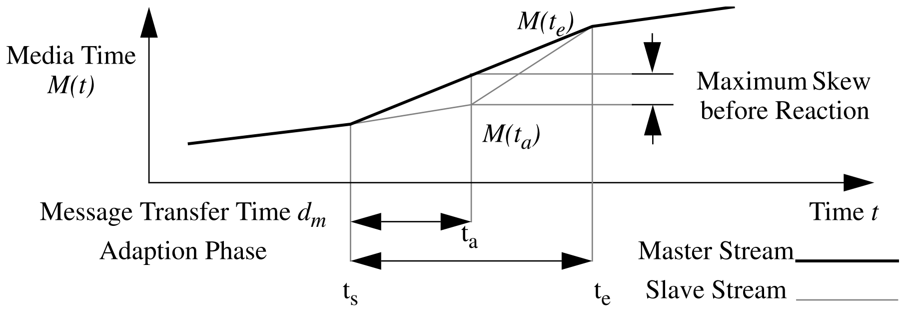

Fig. 3. Master/Slave Synchronization

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 master stream's correction rate R_{corr}: $S_{max} = d_m\cdot|R_{corr}| + \varepsilon$. Between adaption phases, the skew is bounded by ε.

4.5 Master Switching 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 server, 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 server 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*. The 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. An appealing property of our protocol is that a slave can immediately initiate recovery when its stream becomes critical: First, the slave makes a transition to a so-called tentative master (or t-master for short) and informs the server by sending an *IamT-Master* message. Then - without waiting on any response - it enters the adaption phase to move its buffer delay out of the critical region by adapting R_2 accordingly. In order to keep the other streams in sync, it propagates an *Adapt* request to all other sink clients, including the master. At the end of the adaption phase, a t-master falls back in the slave role. Should the stream still be critical by this time, then the recovery procedure is initiated once more.

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 switching 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 client. This requirement can be fulfilled by including a timestamp in *Adapt* requests and performing these requests in timestamp order at the client sites. The latter means that a client 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 on 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: $<E_R.E_M.T>$, 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. 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 server and the clients 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

---

1. We assume that at no point in time there exist two t-masters that try to adapt the release rate in contradicting directions, i.e. one tries to increase the rate while the other tries to decrease it. This is achieved by dimensioning the play-out buffer appropriately.

counter is set to the received epoch value. Moreover, a client increments its local recovery epoch counter when it performs recovery, i.e. the *IamT-Master* message sent to the server already reflects the new recovery period. The server increments its master epoch counter when it selects a new master, i.e. the *GrantMaster* message already indicates the new master epoch.

Adapt requests are accepted only in strict timestamp order. Should a client 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 already mentioned above, a critical slave sends an *IamT-Master* message when it becomes a t-master. When the server 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 just discarded upon arrival (see Fig. 4).

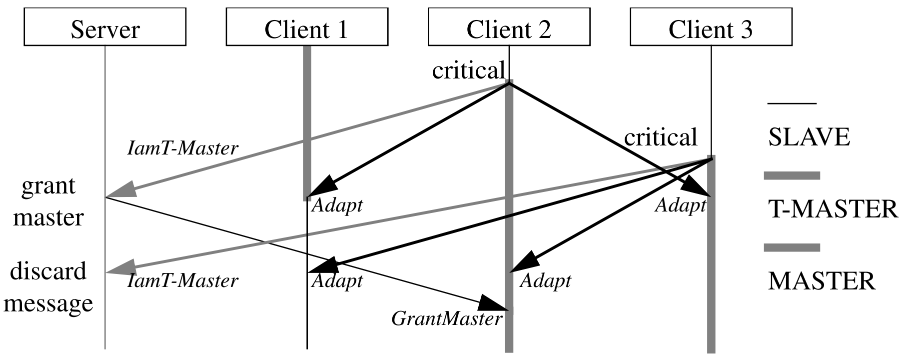

Fig. 4. Recovery-initiated Master Switching

The worst case skew S_{max} among sinks can be observed when master and a t-master decide to adapt their release rates in opposite directions at approximately the same time. S_{max} can be shown to be $d_m \cdot (|R_{corr,\ master}| + |R_{corr,\ t\text{-}master}|) + \varepsilon$, where d_m denotes the transmission delay of *Adapt* messages.

5 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 expected 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 intrastream synchronization quality in terms of data loss due to underflow or overflow is primarily influenced by the *LWM* and *HWM* values. A good rule of thumb for the width of the critical regions defined by these two parameters is *j/2* for each, where *j* denotes the jitter of the corresponding data stream. Increasing *LWM* also increases the quality as the probability of underflow is reduced. On the other hand, this modification may also increase the overall delay, which might be critical for the given application. ASP allows to modify *LWM* and *HWM* values 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 [6] may monitor the data loss rate and adjusts the water marks as needed.

The width of the target buffer area determines the aggressiveness of the buffer control algorithm. The minimum width of this area depends on the smoothing function applied to determine $d_B(t)$. The larger the width of the target area, the less adaptions of the release rate are required. Rather constant release rates require almost no communication overhead for adapting slaves. 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 the *LWM* to minimize the overall delay, a small target area is the better choice.

The location of the target area in the buffer 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 examples, the minimum delay policy and the dedicated master policy.

The goal of the *minimum delay policy* is to achieve the minimum overall delay for a given intrastream synchronization quality. To reach this goal the stream with the currently longest transmission delay is granted the master role, and this stream's buffer delay is kept as close as possible to *LWM*. The target area for the master is located as follows: $LTB = LWM$ and $UTB = LWM + \Delta$, where Δ is the jitter of $d_B(t)$ after smoothing.

Due to changing network conditions it may happen that the transmission 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 server receives an *IamT-Master* message it grants the master role the originator of this message. If it receives multiple *IamT-Master* messages originated in the same recovery epoch only the first one is accepted, all the other ones are discarded. In the long run, this strategy ensures that the stream with the longest transmission delay eventually becomes master. The overall delay at time *t* amounts to the longest transmission delay at *t* plus $LWM+\Delta$, which obviously is the minimal overall delay that can be achieved at *t*.

The possibility of dynamically tuning *LWM* makes this policy very powerful. By increasing the *LWM* value the quality but also the overall delay is increased. Conversely, the quality and delay is decreased if *LWM* is decreased. Consequently, by tuning *LWM* the user may (interactively) determine the appropriate trade-off between delay and intrastream synchronization quality.

The *dedicated master policy* dedicates the master role to a certain stream. This is typically a stream that is played out at a sink device with a fixed consumption rate, such as an audio device. Obviously, the best intrastream synchronization quality for such a stream is achieved if release rate R_2 equals the sink's consumption rate, allowing it to operate without duplicating and skipping. The dedicated master policy ensures that R_2 and the consumption rate may differ only if some stream in the sync group is critical.

During the start-up procedure, the dedicated stream is granted the master role, and the target area is set to its maximum size ($LTB = LWM$ and $UTB = HWM$), reducing the necessity of rate adaptions to a minimum. When a slave becomes critical, it becomes t-master and performs recovery. After recovery, however, the dedicated stream is granted the master role again by the server.

Not only individual parameters but the entire policy can be changed during presentation. For example, the server may start with the dedicated master policy and later on switch to the minimum delay policy when the overall delay gets unacceptable. When changing the policy, the server may require knowledge about the state of the play-out buffers (e.g., current buffer delay, *LWM*, *HWM*). For that purpose, the ASP provides services for requesting buffer state information from clients.

In our opinion, the two synchronization policies described above are the most important ones in practice. However, other policies are conceivable as well.

6 Simulation Results

The section presents some simulation results showing the behavior of ASP's buffer control protocol and discusses results regarding the skew among streams.

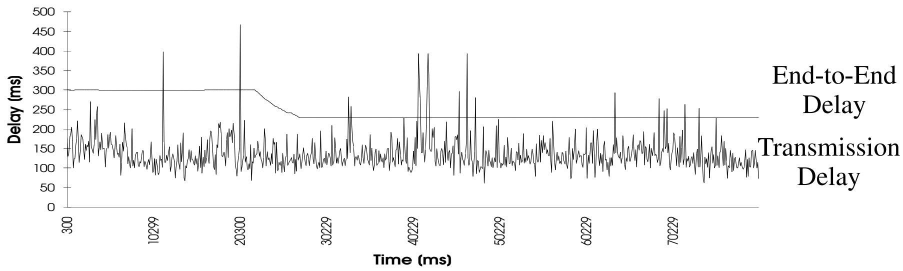

Fig. 5. Transmission and End-to-End Delay

The simulation of the buffer control protocol is based on delay data measured on the Internet. Incoming data units have an average delay between 100 and 250 ms (Fig. 5). The data units are buffered in the play-out buffer of the master stream. Its target area is first set to 100-200 ms (Fig. 6). The buffer delay is smoothed by the geometric weighting smoothing function with α set to 0.9. The buffering leads to a nearly constant release rate R_2 (Fig. 7), which most of the time equals the nominal rate R_1. Data loss due to late arrival amounts to approx. 1% of the overall amount of data units (Fig. 8).

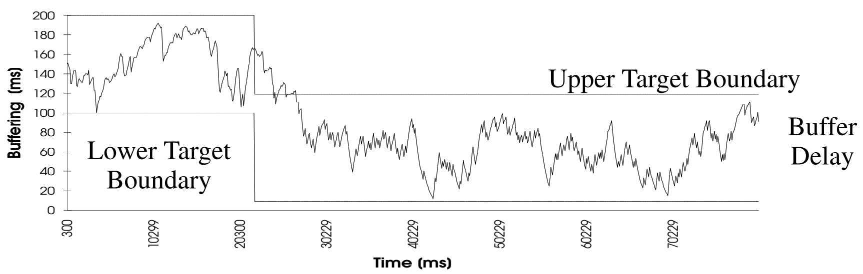

Fig. 6. Buffer Level of Master Stream

As mentioned before, ASP supports the adaption of target levels even when a presentation is in progress. By moving the target area to 20-120 ms, the overall delay of the played out data units can be reduced by approx. 50 ms (Fig. 6). However, the quality of the stream is degraded as the loss rate of data units discarded due to late arrivals is higher (approx. 3% lost data units).

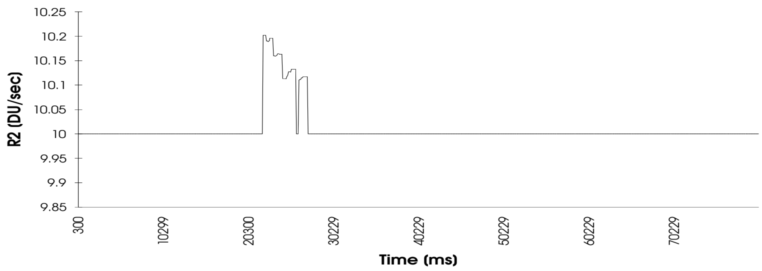

Fig. 7. Release Rate of Master Stream

Our simulations have shown that rate correction R_{corr} is less than 3%. If we assume the delay for control messages to be less than 500 ms, the skew is bounded by 15 ms + ε during an adaption phase (see Sect. 4.4). While the master switching protocol is in progress, the skew bound increases to 30 ms + ε (see Sect. 4.5). Note that during the normal operation (i.e. no master switching and no adaption phase) the skew is bounded by ε, the maximum inaccuracy of clocks.

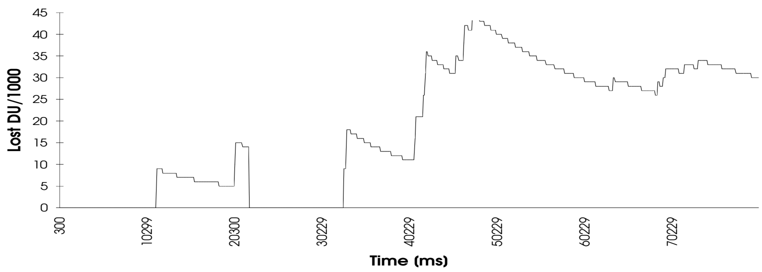

Fig. 8. Losses Due to Late Arrival (Data Units / 1000)

7 Summary

ASP achieves interstream synchronization in distributed environments. It adapts to changing network conditions and allows to tune the quality of data streams to application requirements by supporting a wide range of synchronization policies. Stream quality is improved by reacting on critical situations immediately. Furthermore, by limiting reactions to critical situations, a considerably low message overhead is achieved. The simulation results show good performance even if the underlying communication system does not guarantee quality of service.

The design of ASP was conducted in the context of the *CINEMA* project [12], [13]. *CINEMA* is a system platform for developing distributed multimedia applications. We are currently integrating ASP into the *CINEMA* system. Future work will be to verify our simulation results in the context of various applications.

References

[1] Nipun Agarwal and Sang Son. Synchronization of distributed multimedia data in an application-specific manner. *2nd ACM International Conference on Multimedia, San Francisco, USA*, pages 141–148, 10 1994.

[2] David P. Anderson and George Homsy. Synchronization policies and mechanisms in a continuous media i/o server. *Report No. UCB/CSD 91/617, University of California, Berkeley, CA*, 2 1991.

[3] Andrew Campell, Geoff Coulson, Francisco Garcia, and David Hutchison. A continuous media transport and orchestration service. *SIGCOMM'92 Communications Architectures and Protocols*, pages 99–110, 8 1992.

[4] Julio Escobar, Craig Partridge, and Debra Deutsch. Flow synchronization protocol. *IEEE Transactions on Networking*, 1994.

[5] IBM Corporation. *Multimedia Presentation Manager Programming Reference and Programming Guide 1.0, IBM Form: S41G-2919-00 and S41G-2920-00*, 3 1992.

[6] Thomas Käppner, Falk Henkel, Michael Müller, and Andreas Schröer. Synchronisation in einer verteilten Entwicklungs- und Laufzeitumgebung für multimediale Anwendungen. *Innovationen bei Rechen- und Kommunikationssystemen*, pages 157–164, 1994.

[7] Daniel Köhler and Harald Müller. Multimedia playout synchronization using buffer level control. *2nd International Workshop on Advanced Teleservices and High-Speed Communication Architectures, Heidelberg, Germany*, 9 1994.

[8] David L. Mills. On the accuracy and stability of clocks synchronized by the network time protocol in the internet system. *Computer Communications Review*, pages 65–75, 1990.

[9] Postel. Transmission control protocol, darpa internet program, protocol specification. *RFC 793*, 9 1981.

[10] Srinivas Ramanathan and P. Venkat Rangan. Continuous media synchronization in distributed multimedia systems. *3rd International Workshop on Network and Operating System Support for Digital Audio and Video*, 11 1992.

[11] P. Venkat Rangan, Srinivas Ramanathan, and Thomas Käppner. Performance of inter-media synchronization in distributed and heterogeneous multimedia systems. *Computer Networks and ISDN Systems*, 1993.

[12] Kurt Rothermel, Ingo Barth, and Tobias Helbig. CINEMA - An Architecture for Distributed Multimedia Applications. In *Architecture and Protocols for High-Speed Networks*, pages 253–271. Kluwer Academic Publishers, 1994.

[13] Kurt Rothermel and Tobias Helbig. Clock hierarchies: An abstraction for grouping and controlling media streams. *Technical Report 2/94, University of Stuttgart/IPVR*, 4 1994.