The Concord Algorithm for Synchronization of Networked Multimedia Streams

N. Shivakumar* C. J. Sreenan B. Narendran P. Agrawal

AT&T Bell Laboratories
Murray Hill, NJ 07974

Abstract

Synchronizing different data streams from multiple sources simultaneously at a receiver is one of the basic problems involved in multimedia distributed systems. This requirement stems from the nature of packet based networks which can introduce end-to-end delays that vary both within and across streams. In this paper, we present a new algorithm called Concord, which provides an integrated solution for these single and multiple stream synchronization problems. It is notable because it defines a single framework to deal with both problems, and operates under the influence of parameters which can be supplied by the application involved. In particular, these parameters are used to allow a trade-off between the packet loss rates, total end-to-end delay and skew for each of the streams. For applications like conferencing this is used to reduce delay by determining the minimum buffer delay/size required.

1 Introduction

The use of packet based communication networks to transport digitized streams of audio and video is becoming increasingly common. A characteristic of these networks is that the total delay experienced by each packet is a function of variable delays due to physical media access and relay queueing, in addition to fixed propagation delays. The result is that the time difference between transmitting any two packets at the source is unlikely to be the same as that observed between their arrival at the destination. This is a problem for a stream of multimedia packets, because the presence of delay variations (known as *jitter*) can have an impact on the audio-visual quality as perceived by a human user. Delay is also a problem when using multiple related streams, because it introduces a temporal difference which is undesirable in situations where, for example, *lip-sync* is required. We describe these two problems as single and multiple stream synchronization respectively.

To solve these synchronization problems, one must introduce additional delay by buffering packets at or near the point of presentation. In the single stream case this is used to smooth out jitter prior to passing data samples to an output device. For multiple related streams it is used to compensate for inter-stream delay differences. There are two basic approaches to operating this synchronization buffer (sometimes called an elastic buffer) [5]:

- *Packet-Preserving:* In this scheme, the receiver waits until a packet p arrives to play back the packet.
- *Time-Preserving:* If a packet p does not arrive within a certain time bound, the packet is assumed to be lost, and subsequent packets are played back. If p does arrive later, it is thrown away.

Our focus is on time-preserving applications and the additional requirements associated with the total end-to-end delay: the cumulative delay suffered by packets in the network and buffer. Clearly, if every packet is delayed in the buffer such that it suffers cumulatively (in the network and buffer) a delay equal to the maximum network delay, the receiver can reproduce a jitter-free playback. Problems with this approach are that it may be difficult to obtain an accurate estimate of maximum network delay over a stream lifetime, and such a value may be quite large in relation to the average delay. This is important for real-time applications like conferencing and telephony, where large delays are detrimental to interactivity. For example, in packetized voice applications, figures in the 150 - 250 msec range are often quoted as being the maximum acceptable total end-to-end delay. It is possible, however, to

*Now at the Department of Computer Science, Stanford University.

take advantage of the built in redundancy of a multimedia stream to reduce the buffer delay at the expense of throwing away packets which arrive too late. This may not be the case with other data streams such as file transfers.

In this paper, we present the Concord algorithm, which is notable because it defines a single framework to deal with both forms of synchronization, and operates under the influence of parameters which can be supplied by the application involved. We perform a trade-off of packet loss against the total end-to-end delay suffered by packets, by applying these parameters to probabilistic data describing packet delay distributions. Section 2 gives the details and also describes extensions to deal with the effects of sender/receiver clock drift. In Section 3 the solution for multiple stream synchronization extends this scheme by taking user input describing upper bounds on the acceptable difference in end-to-end delay (known as *skew*) for the related streams. This information is used, if necessary, to increase the selected buffer sizes to achieve synchronization between the streams. The scheme operates dynamically by updating its probability delay distributions and actual buffer sizes over time in response to the observed behavior of the network. Some early simulation work is presented in Section 4; followed by a review of related work and our conclusions in Sections 5 & 6 respectively.

2 Single Stream Solution

This section addresses the problem of single stream synchronization. After giving a formal definition of the problem, our algorithm is described in detail. The initial discussion ignores issues of imperfect clocks, an assumption which is removed and dealt with in Section 2.3.

2.1 Problem Formulation

As shown in Figure 1, we have a network N, over which sender S wishes to send stream M to receiver R, with the packets in M being produced every T_r seconds. All packets have sequence numbers. Stream M is characterized by the *maximum acceptable delay* (MAD) it can suffer, and *maximum packet loss rate* (PLR) it can tolerate. For the network N we construct a *packet delay distribution* (PDD): an estimate of the probable delays suffered by packets in the network over a time window. This PDD may draw on existing traffic conditions, history information or any negotiated service characteristics to derive estimates for minimum, maximum and/or mean delays and delay distributions. The PDD constructed from this information is approximate, but because its long-term accuracy is important, we assume that this can be improved dynamically as described in Section 2.2.5. The basic problem is then to find the minimum buffer size at the receiver that will smooth out network jitter so that the stream's requirements of PLR and MAD are satisfied, if possible. That is, to find B the minimum buffer size satisfying the following:

- For every packet p: $nd_p + bd_p$ is a constant TED, where TED is the *total end-to-end delay*, nd_p is the *network delay* suffered by packet p, and bd_p is the induced *buffer delay* for p, a function of the buffer size.
- The chosen TED is lesser than the MAD of the stream.
- The chosen TED does not lead to more than PLR packets/sec being thrown away.

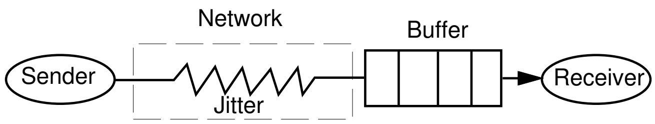

Figure 1: System elements

2.2 Algorithm Details

2.2.1 Computing Total End-to-End Delay

In this section we describe how to calculate the end-to-end delay, using the type of PDD shown in Figure 2 to illustrate the procedure. This PDD shows a normal distribution of packets delays, which is typical of packet switched networks [2]. From this plot it is clear that all packets that suffer more than TED in network delay will be declared lost in the time-preserving scheme. Hence $(1 - Cdf(TED))$ packets will be lost every second, where Cdf is the *cumulative distribution function* on the PDD. There are a few possible directions at this point to choose the TED of the packets.

1. *Minimize PLR:* Choose TED so that PLR is minimized while satisfying the MAD. $Cdf(t)$ is a monotonically increasing function, hence $(1 - Cdf(t))$ is monotonically decreasing. That is, the packet loss rate decreases with an increase in t.

Hence, the choice of $TED = MAD$, will have the minimum packet loss rate, since MAD is the maximum permissible TED.

2. *Minimize TED:* Minimize TED such that the PLR is satisfied. This can be achieved by choosing the minimum TED such that $(1-Cdf(TED))$ is PLR. Choosing any t greater than this TED will reduce the packet loss rate of the stream as seen above. Conversely, any t lesser than TED will lead to a greater packet loss rate than PLR. Hence the given choice of TED will be the minimum t satisfying the given PLR.

3. *Minimizing Hybrid-Cost:* Find a TED such that some cost metric $cost[f(TED, plr)]$ is minimized, where $f(TED, plr) \rightarrow \Re$ with the constraints that $TED \leq MAD$, and $plr \leq PLR$, where plr is $(1 - Cdf(TED))$.

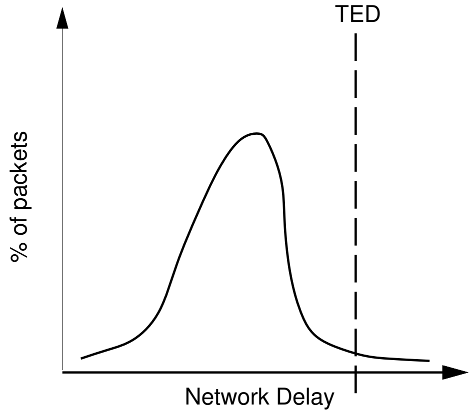

Figure 2: A network delay distribution for packets

When we choose the TED for the packets, we are in effect choosing the buffer delay to be encountered by the packets since network delay is a function of the network. It is clear that TED can be chosen satisfying both the MAD and PLR constraints iff $(1 - Cdf(MAD)) \leq PLR$.

2.2.2 Operation of the Buffer Controller (BC)

Conceptually, the buffer for delaying packets is viewed as a simple *shift-register* mechanism[1]. Every T_r seconds, packets in the i^{th} slot, are moved to the $(i-1)^{th}$ slot for all $i = 1, 2 \ldots B$, where T_r is the rate at which R replays packets (and at which the S produces packets), and B is the buffer size. The packet in the zeroth slot, henceforth termed the *EXIT* slot will be given to the receiver also every T_r seconds (see Figure 3). The buffer controller uses two variables - *PLAYED* to indicate which packet is to be played next, and *TIMER* which can be initialized and reset to keep track of T_r. In this, we allow a network to resequence and duplicate packets.

[1]A practical implementation avoids copying packet contents by manipulating pointers to packets or using a circular buffer.

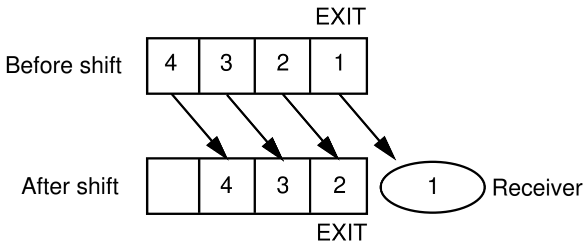

Figure 3: Buffer mechanism

1. Given the initial PDD of the network, and the PLR, MAD of the stream and the cost function to minimize, BC will decide on the TED.

2. When the first packet arrives (with sequence number i), the nd_i suffered by the packet is calculated, and the packet is placed into the appropriate buffer slot as defined by $b_i = \lfloor (TED - nd_i)/T_r \rfloor$. Set $PLAYED = (i - b_i)$, and $TIMER = T_r - ((TED - nd_i) \bmod T_r)$. Note that i could be any packet, and not necessarily the first packet.

3. When $TIMER = T_r$ seconds, BC will pass the packet in the *EXIT* slot to the receiver, will increment *PLAYED* by one, and will shift all enqueued packets by one. This is a discretely timed event. Reset $TIMER$ to zero.

4. When a new packet arrives with sequence number j, the BC throws away the packet if j is less than the current value of *PLAYED*. Otherwise, the packet is placed into the $(j-PLAYED)^{th}$ slot in the buffer. Observe that by the shifting mechanism, packet j will continue to remain in the $(j-PLAYED)^{th}$ slot even as $PLAYED$ changes. And since all distinct packets have unique sequence numbers, no two such packets will collide in the buffer. Duplicated packets do not create any problems, since the slot will be occupied by exactly one instance of the packet.

Lemma 1 *The first packet with sequence number i that arrives at the receiver with nd_i network delay, $nd_i \leq TED$, would suffer exactly TED from being produced to being serviced at the receiver.*

Proof of Lemma 1 : By construction i will be placed into slot $b_i = \lfloor (TED - nd_i)/T_r \rfloor$. After $((TED - nd_i) \bmod T_r)$, the $TIMER$ would go off, and the packet will be shifted to the right by one. The packet will take exactly $T_r * b_i$ from then to be serviced. Hence, the packet would suffer a cumulative delay of

$$nd_i + ((TED - nd_i) \bmod T_r) + T_r * (\lfloor (TED - nd_i)/T_r \rfloor)$$

$= nd_i + (TED - nd_i)$, since $\lfloor (p/q) \rfloor * q + p \bmod q = p$.

Lemma 2 *Subsequent packets that arrive with higher or lower sequence numbers than the first packet i received, will be played back if their $nd \leq TED$, or else will be thrown away.*

Proof of Lemma 2 : Let i, the first packet received, be produced at time t_i. Hence, it would be played at time $t_i + TED$, by Lemma 1. At $t_i + TED$, *PLAYED* would be set to $i + 1$. Consider a packet $(i + j)$ that arrives at the buffer, where j could be positive or negative. It must have been produced at time $(t_i + j * T_r)$. $(i + j)$ would lose its chance of being played if it arrived when *PLAYED* $> (i + j)$. This would happen when time is greater than $(t_i + TED + j * T_r)$, by construction. Hence if $(i + j)$ suffers $nd <= TED$, $(i + j)$ would reach the buffer before the deadline of $(t_i + TED + j * T_r)$ and will be played, else it will be thrown away. Note that we are not using the explicit value of the network delay suffered by subsequent packets, instead relying on sequence numbers to derive timing information.

Theorem 1 *Packets that suffer $nd \leq TED$ will be played such that they suffer exactly a cumulative delay of TED.*

Proof of Theorem 1 : Let i be the first packet that arrived. If it was produced at t_i, from Lemma 1, we see that the first packet arrived at $t_i + TED$. Any packet $(i + j)$ will be produced at $t_i + T_r * j$, and would lose its chance of being played back $T_r * j$ after i was played. Hence every packet that arrives within TED, will suffer a cumulative delay of $(t_i + TED + j * T_r) - (t_i + j * T_r)$, the difference between the time of service and production, which is TED.

2.2.3 Computing Buffer Size

Finding the minimum buffer size required can be alternatively posed as finding the maximum number of buffer slots that could be occupied at a given time. Let us say that the first packet received had sequence number 1 at time 0. The worst case occurs when the k^{th} packet arrives at the receiver's buffer after having suffered minimum network delay min, at time $((k-1) * t_r + min)$. The first packet would have been serviced at time TED by the receiver. The maximum number of packets in the buffer is

number produced $-$ number serviced $=$

$$k - \lfloor \frac{(((k-1)*T_r + min) - TED)}{T_r} \rfloor = -\lfloor \frac{(min - TED - T_r)}{T_r} \rfloor.$$

The above relation can be proved with a little more effort to any packet reaching the buffer first, if one should consider the relation between *PLAYED* and the first packet inserted, and consider the buffer to be a circular buffer for analytical reasons. An alternative proof is from the fact that all packets have to suffer the maximum delay in the buffer if they suffered the minimum delay in the network so that the cumulative delay is TED.

2.2.4 Computing Network Delays

In some networks, link speed between two nodes in bytes/second is typically known. This defines the minimum amount of propagation time that will be needed for a packet of $size_p$ bytes to be transmitted. That is $min = size_p/link$ secs independent of switching/ buffering etc. Note that this information is not critical. If the link speed cannot be computed, min can be set to zero for the computation of buffer size. However, if min is known in the network, the buffer size can be optimized with this added knowledge.

The algorithm proposed depends on knowing the network delay suffered by the first arriving packet. A review of possible approaches to computing the network delay of packets is given in [7]. One common approach is to keep the source and the receiver clocks synchronized[2]. A time-stamp on the packets when they are produced, will allow us to compute the network delay suffered by the packet. Another approach is to have the packet switches record propagation and buffer delays in the packets. Note that none of the lemmas/ theorem depend on knowledge of network delays for packets received after the first packet, because

[2]Distributed clock synchronization schemes such as [6] can maintain accuracy to within milliseconds.

of the use of sequence numbers to derive the timing. Hence certain optimizations can be made in case of more knowledge of the network, and packets after the first received packet need not carry the extra timing information, or have time stamps.

If one of the schemes in [7], cannot be implemented, and clock synchronization algorithms are not feasible we cannot obtain the network delay suffered by the first packet. Hence, we cannot guarantee the optimality of our algorithm since the first slot of insertion cannot be obtained implicitly. One alternative approach considered is to use a **Special Control Packet** . If the network can guarantee that the packet is delivered between $[Min, Min + \lambda]$, then

- if $\lambda \leq T_r$ insert the first packet into the buffer slot, denoted by **END** that will delay the packet to the maximum extent. Hence the TED suffered by the packets is actually bounded by $TED_{optimal} + \lambda$ or
- if the stream is sequenced, and $\lambda > T_r$, insert the first packet into **END**, and whenever a new packet i arrives, insert it into the appropriate slot offset from the $PLAYED$ packet. If the slot computed is higher than the size of the buffer by k, shift all the packets to their right by k slots so that the new packet is accommodated. Some packets may drop off the left edge. This is because they have suffered excessive delay relative to the new packet. In the beginning, the actual total end-to-end delay TED_{beg} is bounded by $TED_{optimal} + \lambda$. Over time, this can be reduced by the throwing away of packets based on the arrival pattern of packets as in [2].

2.2.5 Dynamic Operation

As mentioned in Section 2.1, an initial estimate of the network PDD is constructed based on information such as history and/or the prevalent traffic conditions at the time of stream setup. But over time, the conditions of the network and the delays suffered by the packets can change. If the PDD of the network changes, the assertions on the computed TED and the buffer sizes will no longer be valid. Hence we need some mechanism to update the PDD of the system.

It is trivial to compute the PDD of the network if the network delays of the packets are known. Since we do not assume time stamps in our packets, we do not have an explicit value for the network delays. In this section, we will introduce a scheme to compute the network delays at the receiver end. Let us say that the first packet is serviced at t_1, at the local clock of the receiver.

1. When a packet p is inserted into the buffer, time-stamp the packet with the clock $time_{arrive}$ at the receiver. When it is being removed for service at $time_{svc}$, compute the buffer delay bd_p to be $time_{svc} - time_{arrive}$. Since by Theorem 1, packet p should suffer a cumulative delay of TED, we have the nd_p suffered by the packet to be $TED - (time_{svc} - time_{arrive})$.
2. If the i^{th} packet arrives after TED, by Lemma 2, the packet will not be inserted into the buffer since the packet's service time would have elapsed at time $t_1 + (i-1) * T_r$. By computing $time_{extra}$ to be $time_{arrive} - t_1 + (i-1) * T_r$ for the i^{th} packet, we see that the i^{th} packet has suffered exactly $TED + time_{extra}$ in the network.

It should be noted that all the above times are by the local clock of the receiver. The above scheme helps us maintain the PDD of the network over any required time window without any additional packet information or undue computation. The user application can be given access to this data, and can respond to increase/ decrease buffer size based on the application's requirements.

2.3 Dealing with Imperfect Clocks

2.3.1 Clock Offset and Drift

Note that we do not rely on the absolute value of the clocks at the source and the receiver in our buffer controller mechanism. However, if we do choose to use time-stamps to compute the network delay of the first packet as opposed to other methods in [7], we will need to worry about clock offsets. Let us say that the clock synchronization algorithms executed guarantee that the source and receiver's clocks are never off by more than δ. Hence the real network delay nd_{true} of the first packet is within $[nd_{computed} - \delta, nd_{computed} + \delta]$, where $nd_{computed}$ is the network delay computed from the time-stamp. Choose $TED_{operation}$, the intended total end-to-end delay in the range $[TED_{optimal} - \delta, TED_{optimal} + \delta]$ depending on the PLR/TED trade-off metric of the user application, where $TED_{optimal}$ is the optimal TED that would have been chosen in absence of the δ-uncertainty. For instance, if one still wants to ensure the PLR is maintained, choose $TED_{operation}$ to be $TED_{optimal} + \delta$ and when the first packet arrives, assume the packet suffered the best case delay of $(nd_{computed} - \delta)$, and perform the appropriate buffer initializations. Better bounds on δ

will of course ensure increased accuracy of the computed network delay of the first packet. Since none of the lemmas/ theorem need the network delay information for subsequent packets, even if the time-stamping clocks do drift over time, the buffer mechanism is not affected.

2.3.2 Drifting Production and Service Rates

Typically, there are clocks associated with the device generating the packets and the receiver that processes the packets. The buffer controller algorithm presented thus far depends on the device clocks of the source and receiver not drifting. That is, rate of production and service is assumed to be the same and to remain constant. In practice, physical clocks may operate at slightly different rates, which over time can lead to buffer overrun or overflow at the receiver and cause a loss of synchronization. In this section, we make a relaxation to allow for device clock drifts, and describe a technique which makes infrequent adjustments to stream delay at the receiver in order to compensate for drift effects. Observe that a clock drift of sender or receiver can be considered a drift of the sender relative to the receiver. Hence we can work with the assumption that the receiver does not drift. Let T_s and T_r be the device period of the source and receiver, that is the rate of production and service respectively.

- **Source faster than Receiver:** If this scenario is left unchecked, the source will flood the receiver and the buffer will overflow. Consider the i^{th} packet. The buffer size can be exceeded iff the buffer is already full, and the i^{th} packet arrives before any packet is serviced. The worst case is when the i^{th} packet reaches the buffer at time $i * T_s + min$. Due to the drift, if the source's device is at time t, the receiver's clock will be at time $(t * T_s)/T_r$. Similar to the section on computing the buffer size, we see that the buffer size is actually

  maxbufsize $= i - \lfloor \frac{(i*T_s+min-TED)*T_s}{T_r^2} \rfloor$.

  Clearly, the above depends on i, the number of packets sent. However, for real-time applications, an advance bound cannot usually be obtained on the length of the conversation. One possible approach is to throw away a packet every p secs so as to maintain the bound on the number of buffer slots required. Therefore, the new maxbufsize
  $= i - \lfloor \frac{(i*T_s+min-TED)*T_s}{T_r^2} \rfloor - \lfloor \frac{(i*T_s+min-TED)*T_s}{T_r*p} \rfloor$
  $\leq i - \frac{(i*T_s+min-TED)*T_s}{T_r^2} - \frac{(i*T_s+min-TED)*T_s}{T_r*p} + 2$.

  Consider only the factor that depends on i, which is $i * \frac{T_r^2*p - T_s^2*p - T_s^2*T_r}{T_r^2*p}$.

  The easiest way to make the buffer size to be independent of i, is to set the above term to zero, and solve p to be $(T_s^2 * T_r)/(T_r^2 - T_s^2)$. Hence throwing one packet away every p secs, ensures that the buffer size is bounded.

- **Source Slower than Receiver:** In this case, the receiver services the packets faster than the source produces. The receiver will be ready to service the i^{th} packet at time $t_i = TED + (i-1) * T_r$ after the first packet was produced. The number of packets that would have been produced by time t_i would be $(t_i * T_r/T_s^2)$. A packet should have left the source TED before its due time at receiver. Hence the number of packets that would reach the $EXIT$ slot in the buffer by t_i is $packet_{arrive}$ $= \lfloor \frac{((t_i - TED)*T_r)}{T_s^2} \rfloor$.

  We need to have $(packet_{arrive} - i) \geq$ zero so that the receiver can always process a packet. Again we see that the difference depends on i, and we do not have bounds on the length of a multimedia conversation. One possibility is to add a dummy packet, or make a copy of another packet every p secs so that the source- receiver relation is maintained such that

  $$\lfloor \frac{(t_i - TED) * T_r}{T_s^2} \rfloor - i + \lfloor \frac{(t_i - TED) * T_r}{T_s * p} \rfloor \geq 0.$$

  We can make the above equation independent of i, when p is $(T_r^2 * T_s)/(T_s^2 - T_r^2)$. So every p secs, we insert an extra packet into the buffer.

In devices and applications, where the exact drifts are known, we can compute the exact rate at which packets can be thrown or dummy packets be inserted. If the exact drift is not known, but the drift can be bounded, we can resort to choosing the earliest time obtained by computing the times associated with the various combinations of direction of drift and maximum drift in the two cases. The buffer controller can then have an estimate on when to check the buffer, and depending on how full the buffer is, the user application will have the flexibility to insert or throw away a packet.

3 Multiple Stream Extension

In the previous section, we considered how to compute the total end-to-end delay and the buffer size for a single stream case. In this section, we consider the multi-stream case where N streams of data are being sent over different links from multiple sources S_i, i = 1, 2 ... N, to a common receiver R. Computing the TED for a set of streams is similar, but more involved due to the PLR_i and MAD_i requirements for each of the N streams. We will first consider a simple case of synchronizing the N streams such that:

- TED is the same for all the N streams.
- $TED \leq MAD_i$ for $i = 1, 2 \ldots$ N.
- The packet loss rate for stream i is bounded by PLR_i, for $i = 1, 2, \ldots$ N.
- TED is the minimum t satisfying the above conditions. (Other metrics defined in section 3.1 are also clearly possible.)

3.1 Computing Operational Parameters

We now determine the system operation metrics according to the following steps:

- As before, calculate each stream delay $t_i = Cdf_i^{-1}(1 - PLR_i)$ for each i = 1, 2, ..., N.
- Choose TED to be the maximum of the computed t_i, i = 1, 2 ... N, hence satisfying all PLR requirements.
- If $TED > MAD_i$ for any i = 1, 2, ... N, there is no feasible solution for these stream parameters.
- Choose B_i, the buffer sizes for the N streams to be $(TED - min_i)/T_r$, where min_i is the minimum network delay for sources S_i, i = 1, 2, ... N.

3.2 Relaxing the Delay Constraints

In some applications, streams have less strict constraints regarding their synchronization. For example, audio streams may be required to be within 100 msec of related video streams. Using this extra degree of freedom, some otherwise infeasible requirements can be satisfied. We consider the case where the TEDs between any two streams are permitted to be at most α apart:

- Compute $t_i = Cdf_i^{-1}(1 - PLR_i)$ for the i^{th} stream as in the previous section.
- The constraints are infeasible if $\max t_i - \alpha > \min MAD_i$.
- Otherwise, choose a $t \in (\max t_i - \alpha, \min MAD_i)$ and $TED_i \in (t, t + \alpha) \cap (t_i, MAD_i)$, so as to minimize the desired cost metric.
- Compute buffer size B_i of stream i to be $(TED_i - min_i)/T_r$.

3.3 Semantics of Streams

The TED should be chosen depending on the relation between streams since this will give us more information about what choices are allowable to improve the synchronization mechanism.

- **Master-Slave Streams:** In some cases, one stream is designated the *master* stream, and the rest of the streams, termed the *slaves*, are maintained in synchronization with the master. In this scheme, all received packets of slaves are thrown away if the master's packet is not received. If the master packet is received, then all the available slave packets are played with the master packet. The PLR_{system} of the system is typically specified. This is merely a special case of the system described in section 4 with the PLR_{master} being the PLR_{system}, and the PLR_{slave} being set to 100%.
- **Inter-Dependent Streams:** Sometimes, the streams are related such that even if a packet on one stream is late, the corresponding packets in other streams are to be thrown away in an *all-or-nothing* format. The PLR_{system} is then specified to be satisfied. The joint cumulative distribution function of the $PDD's$ of the streams will then be useful. For instance, if there are n streams X_1, X_2, ..., X_n that need to be synchronized in this fashion.
  - Compute $Cdf(t)$ to be $J_{X_1 \leq t, X_2 \leq t, \ldots X_n \leq t}$ which specifies the probability that the packets of all streams arrives within some time t. Hence $(1 - Cdf(t))$ specifies the probability that packets are to be thrown away.
  - Find TED to be minimum t that satisfies $(1 - Cdf(t)) \leq PLR_{system}$.
- **Redundant Streams:** Critical transmissions are sometimes replicated and transmitted so that the receiver can mix the streams by some sort of a *majority* scheme. In this case the BC needs to choose TED to be the minimum t satisfying $J_{X_1 > t, X_2 > t, \ldots X_n > t} \leq PLR_{system}$ since the computed joint probability defines the probability

that none of the packets are received in time to be used for playing back.

- **Hybrid Streams:** Several streams having hybrid relations can also be specified in this scheme. For example, if there are n sets of streams, and there are m_i streams in the i^{th} set that follow a master-slave relation, while the sets themselves are inter-dependent or redundant, we can find the TED as follows.
  - For the i^{th} set, set the PDD_i to be that of the master stream. $i = 1, 2, \ldots n$.
  - Compute the TED with these new PDD_i as defined in the inter-dependent or redundant scheme.

  Other combinations can be addressed similarly.

In multimedia synchronization, related streams can have different rates of production. For instance, in video-conferencing four audio packets may be generated for every video frame produced. This relation can be modeled at the receiver's end that four audio packets are to be serviced for every video frame serviced.

Let us consider the lip-syncing example where α audio packets are to be serviced for every video packet. Modify the buffer size computing algorithm as follows.

- Compute the TED based on the semantic relation between audio-video.
- Compute the size of the audio stream's buffer to be $(TED - Min_{audio})/T_{audio}$.
- Compute the size of the video stream's buffer to be $(TED - Min_{video})/\alpha * T_{audio}$.

4 Simulation Results

This section presents some early simulation results of Concord based on a system with the following characteristics:

- Three streams are to be synchronized perfectly. That is, they share the same TED.
- The three streams are sent over different links from different sources.
- The packet delay distributions for each of the links followed a normal distribution with means of 100, 90, 100 msecs. The respective variances were 10, 5, and 5 msecs. The minimum link speeds were assumed to be 5, 10, and 5 msecs.
- The rate of packet production, and packet service were assumed to be one packet every 33 msecs.
- Each of the streams were assumed to have a maximum PLR to be 5%, and their maximum acceptable delays were 250 msecs.

We show below the most interesting plots for the cases when the variance, packet loss rates and the rate of production were varied.

- In Figure 4, we varied the rate of service and production for the three packets from 1 msec through 100 msecs, while fixing the rest of the parameters. We plot the buffer sizes required to service the packets at the given rate. As expected, the buffer sizes are maximum when the rate of production is minimum.
- In Figure 5, we varied the max PLR of the third stream from 1% through 15%. We plot the change in TED for the given PLR's. The TED is maximum for the low PLR_3, and beyond the 5% mark, stays constant since the other two streams have maximum acceptable PLRs of 5%.
- In Figure 6, we change the variance of the third stream from 1 msec through 25 msecs. As expected the TED increases beyond the variance of 5 msec. This is because until that point, the variance of the third stream is dominated by the first and second streams. Beyond the 5 msecs point, the third stream is the dominant factor.

5 Related Work

Early work on packetized voice studied the use of a receiver buffer to smooth jitter [3, 7]. A number of experimental systems such as [1, 10], used a time-preserving scheme. More recent work has produced a scheme for calculating accurate delay estimates [2] in the absence of global clocks, and compared policies for changing the buffer size in packet-preserving operation [8].

For the most part, work on multiple stream synchronization has been looked at separately from the single stream requirements. Some proposals, such as [4] assume that reasonable upper bounds on the delay across streams are available, while other schemes set a fixed upper delay bound for a stream, discarding data which arrives late and would cause a loss of synchronization [11]. Other efforts have looked at describing synchronization using some form of data

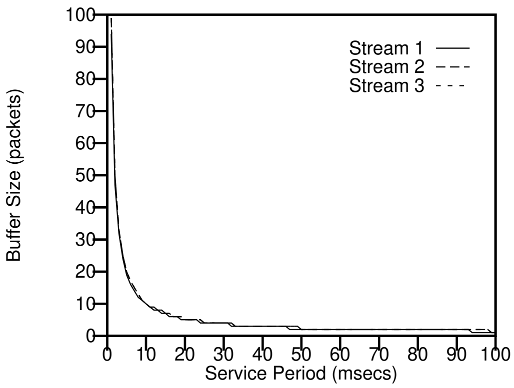

Figure 4: Effect of changing the rate of service

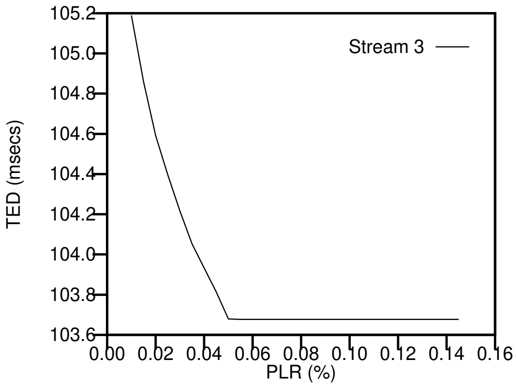

Figure 5: Effect of changing the PLR

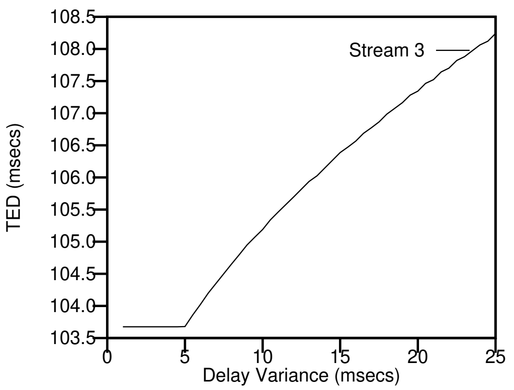

Figure 6: Effect of changing the delay variance

frames, relying on the underlying transport protocol to constrain jitter and delay [5, 9]. The added complications of dealing with streams from different senders was discussed recently in [12] as the inter-participant synchronization problem.

6 Conclusion

This paper presented the Concord algorithm for synchronizing networked multimedia streams. It is notable because it defines a common framework to deal with both single and multiple stream synchronization, and operates under the influence of parameters which can be supplied by the application involved. In particular, these parameters are used to allow a trade-off between the packet loss rates, total end-to-end delay and skew. Thus, an application can directly indicate an acceptable lost packet rate, rather than having the synchronization mechanism operate by always trying to minimize losses.

We have described the details of the basic algorithm and its extension for dealing with multiple stream synchronization. Issues of calculating the receiver buffer size and estimating network delays were discussed. In addition, features for dealing with imperfect clocks and dynamic network behavior were described. Some early simulation results were presented; future work will include a more extensive evaluation with real traffic traces.

References

[1] S. Ades et al. *Protocols for Real Time Voice Communication on a Packet Local Network.* In Proceedings of IEEE ICC '86, pages 525–530, June 1986.

[2] F. Alvarez-Cuevas et al. *Voice Synchronization in Packet Switching Networks.* IEEE Network, pages 20–25, September 1993.

[3] G. Barberis and D. Pazzaglia. *Analysis and Optimal Design of a Packet-Voice Receiver.* IEEE Transactions on Communications, 28(2):217–227, February 1980.

[4] J. Escobar et al. *Flow Synchronization Protocol.* In Proceedings GlobeCom '92, pages 1381–1387, 1992.

[5] L. Li et al. *Synchronization in Real Time Multimedia Data delivery.* In Proceedings of IEEE ICC '92, pages 587–591, June 1992.

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

[7] W. A. Montgomery. *Techniques for Packet Voice Synchronisation.* IEEE Journal on Selected Areas in Communications, 1(6):1022–1028, December 1983.

[8] R. Ramjee et al. *Adaptive Playout Mechanisms for Packetized Audio Applications in Wide-Area Networks.* In Proceedings of IEEE INFOCOM '94, pages 680–688, June 1994.

[9] K. Ravindran and V. Bansal. *Delay Compensation Protocols for Synchronization of Multimedia Data Streams.* IEEE Transactions on Knowledge and Data Engineering, 5(4):574–589, August 1993.

[10] D. C. Swinehart et al. *Adding Voice to an Office Computer Network.* In Proceedings of IEEE GlobeCom '83, November 1983. Also available as Xerox PARC technical report CSL-83-8, February 1983.

[11] R. Yavatkar. *MCP: A Protocol for Coordination and Temporal Synchronization in Multimedia Collaborative Applications.* In Proceedings of IEEE International Conference on Distributed Computing Systems, pages 606–613, June 1992.

[12] P. N. Zarros. *Statistical Synchronization Among Participants in Real-Time Multimedia Conference.* In Proceedings of IEEE INFOCOM '94, pages 912–919, June 1994.