A Realtime Software Solution for Resynchronizing Filtered MPEG2 Transport Stream*

[Extended Abstract]

Bin Yu, Klara Nahrstedt
Department of Computer Science
University of Illinois at Urbana-Champaign
binyu,klara@cs.uiuc.edu

ABSTRACT

With the increasing demand and popularity of multimedia streaming applications over the current Internet, manipulating MPEG streams in a realtime software manner is gaining more and more importance. In this work, we studied the resynchronization problem that arises when a gateway changes the data content carried in an MPEG2 Transport stream. In short, the distance between original timestamps is changed non-uniformly, and decoders will fail to reconstruct the encoding clock from the resulting stream. We propose a cheap software realtime approach to solve this problem. Experimental results from a realtime HDTV stream filter shows that our approach is correct and efficient.

1. INTRODUCTION

With the increasing demand and popularity of multimedia streaming applications and the ever-growing scale and heterogeneity of the Internet, adaptive content delivery, especially with intermediary proxy support has become an extensively studied area. Examples include ProxyNet [1], IBM Transcoding proxy [2], UC-Berkeley TranSend [3] and Content Service Network [4]. One typical service these proxies may provide is to "filter" the media stream , that is, to transform it by changing the content or adding new values to it so that the resulting stream is better matched with resource availability and user preference. Example filter operations include Frame Size Reduction, Low Pass Filtering [5], and Information Embedding [6].

At the same time, MPEG2 [7] has become the most accepted standard for video storage and transmission. MPEG2 system layer stream, especially Transport Layer Stream, is adopted widely for video distribution applications like digital TV broadcast, since it provides many realtime functionalities such as clock synchronization between encoder and decoder, decoding/presentation timestamping and multiplexing of multiple streams with different time base. However, when it comes to HDTV [8, 9], its huge amount of data volumn literally disables any software encoding/decoding approach yet, as far as we know, almost all hardware encoder/decoder boards from the industry only deal with Transport Layer MPEG2 format.

Therefore, despite the importance of MPEG2 stream filtering, not much effort has been done in this area, and many problems remain to be solved. In this paper, we discuss one major problem that has to be solved before all further attempts can be made. To be more specific, we have found that editing or filtering of the HDTV (High Definition TV) stream will cause the access units to drift from their original time point within the timeline carried by the stream, which will hinder, or even disable the decoder from decoding and presenting the video content in a timely manner. Our goal is to identify the cause of the problem for MPEG2 Transport stream and give an efficient realtime software solution.

This paper is organized as follows: In section 2, we will briefly introduce how the synchronization between MPEG encoder and decoder works and the problem that arises after the filtering operations. Our solution is then discussed in detail in section 3 and experiment results follow in section 4. Finally we conclude this paper in section 5.

2. THE SYNCHRONIZATION PROBLEM

Figure 1 (on the next page) shows the MPEG2 Transport stream's management to maintain synchronization between the sender, which encodes the stream, and the receiver, which decodes it. As the elementary streams carrying video and audio content are packetized, their target Decoding Time Stamp (DTS) and Presentation Time Stamp (PTS) are determined based on the current sender clock and inserted into the packet headers. For video streams, the access unit is a frame, and both DTS and PTS are given only for the first bit after the picture header of every frame, which are later used by the decoder to control the speed at which it starts to do decoding and presentation. For example, if at time 5.0s an encoded frame comes to the multiplexing stage, and the encoder believes that the decoder should begin to decode that frame 0.5s after it receives it and output the decoded frame 0.1s thereafter, then the DTS should be set to 5.5s and the PTS 5.6s. After that, as all of these packetized elementary stream packets are further multiplexed together, the final stream is time-stamped with Program Clock Reference(PCR), which is given by periodically sampling the sender clock. This result transport layer stream is then sent over the network to the receiver, or stored in storage devices for the decoder to read in the future. As long as the delay the whole stream experiences remains constant from the receiver's point of view, the receiver should be able to reconstruct the sender's clock that has been used when the

*

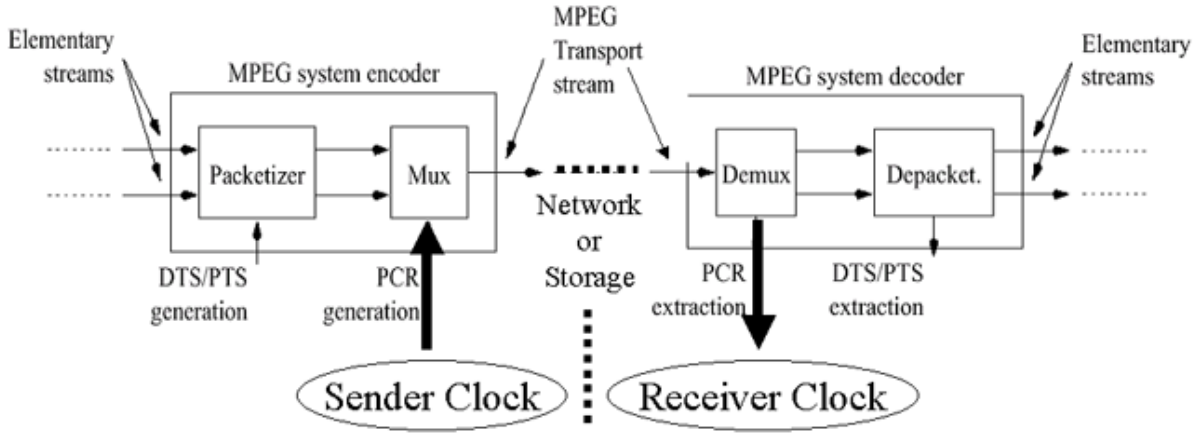

Figure 1: Synchronization between the encoder and decoder

stream was encoded. The accuracy and stability of this recovered clock is very important, since the decoder will try to match the PTS and DTS against this clock to guide its decoding and displaying activities.

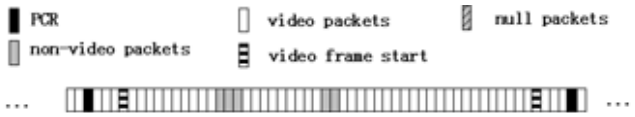

Figure 2: MPEG2 Transport Stream Syntax

Knowing the general idea in timing, we now introduce how the Transport Layer syntax works as shown in Figure 2. All substreams (video, audio, data and timestamps) are segmented into small packets of constant size (188 bytes), and the Packet ID(PID) field in the 4-byte header of a packet tells which substream that packet belongs to. The PCR packets occur at constant intervals, and they form a running timeline along which all other packets are positioned at the right time point. Data packets arrive and are read into the decoder buffer at constant rate, and this rate can be calculated by dividing the number of bits between any 2 consecutive PCR packets by the time difference between their time stamps. In other words, if the number of packets between any 2 PCR packets remains constant, then the difference between their time stamps should also be constant. In an ideal state, packets are read into the decoder at the constant bitrate, and whenever a new PCR packet arrives, its time stamp should match exactly with the receiver clock, which confirms the decoder that so far it has successfully reconstructed the same clock as the encoder. However, since PCR packets may have experienced jitter in network transmission or storage device accessing before they arrive at the receiver, we can not simply set the receiver's local clock to be the same as the time stamp carried by the next incoming PCR no matter when it comes. To smooth out the jitter and maintain a stable clock with a limited buffer size at the receiver, generally the receiver will resort to some smoothing technique like the Phase-Locked-Loop(PLL) [10] to generate a stable clock from the jittered PCR packets. PLL is a feedback loop that uses an external signal(the incoming PCR packets in our case) to tune a local signal source(generated by a local oscillator in our case) to generate a relatively more stable result signal(the receiver's reconstructed local clock in our case). So long as the timing relation between PCR packets is correct, the jitter can be smoothed out with PLL.

After the brief introduction on the usage and importance of the PCR packets, now we are ready to discuss how the filtering operation may affect their validity and accuracy. First, even for the same type of filtering operation, e.g., low pass filtering, for different frames, the time required to do the calculation and processing can be quite different. Since the filtering is transparent to the decoder, it seems to the decoder that the jitter the stream has experienced is larger. This is not a big problem, since through longer buffering at the filter and the receiver and the use of jitter smoothing mechanisms, this stronger jitter will not greatly affect the decoding process. The second problem, however, is more intractable. As we said above, the packets for any access unit should be positioned within the stream and so arrive at the receiver at its supposed time point for the decoder to schedule where and how long to buffer it before decoding

it. However, normally after the filter operations, a video frame becomes smaller or larger. It takes less or more packets to carry, and so its following frames are dragged earlier or pushed later along the timeline. In such circumstances, if we keep both the time stamp and the spacing of the PCR packets unchanged, then the receiver's clock can still be correctly reconstructed, but the arriving time of each frame will be skewed along the timeline. For example, if the stream is low pass filtered, then every frame becomes shorter, and so following frames are dragged forward to pack up the vacancy spared out. From the decoder's point of view, more and more future frames begin to come earlier and earlier, and to buffer them until their stamped time for decoding, the buffer will be overflowed in the long run no matter how large it is. The fundamental problem is that after the filtering, the actual bitrate becomes lower or higher, but the data is still read in by the decoder at the original rate. So if the new rate is lower, more and more future data is read in by the decoder, causing the decoder buffer to overflow eventually; on the other hand, if the new rate is higher, then at some point in the future, the data will be read in after its decoding time has already passed.

3. OUR SOLUTION

One immediate thought would be to do the same kind of clock reconstruction as the decoder does at the filter, and so re-generating the PCR packets to reflect the changes. However, we know that the smoothing mechanisms like PLL are implemented in hardware circuits containing a voltage controlled oscillator that generates high frequency signals to be tuned with the incoming PCR time stamps. But this is not easy, if not impossible, to be done in software. We imagine that a working software version of this scheme require some special realtime support from the operating system kernel, which we have not fully explored yet. Further more, a software filter is a much more convenient form for service distribution and proxy propagation over a wide area network, such as the Internet, because the only resource required is CPU and memory and not any special purpose hardware.

The key idea of our solution comes from the observation that if we do not take an atomic view of each frame, then the DTS and PTS are only associated with the beginning bit of each frame. Consequently, so long as we can fix that point to the correct position on the time line, the decoder should be working fine even if the remaining bits of that frame following the starting point stretches shorter or longer.

3.1 Simple Solution: Padding

Following the discussion above, we have designed a simple solution that works for bitrate reducing operations. We do not change the timestamp and the position of any PCR packet along the timeline within the stream, and we also preserve the position of the frame header and so that of the beginning bit of every frame. What is changed here is the size of each frame in terms of number of bits, and we just pack the filtered bits of a picture closely following the picture header. Since each frame takes less packets to carry, yet the frame headers are still positioned at their original time points, we can imagine that there would be some "white space" left between the last bit of one frame and the first bit of the header of the next frame. Actually the capacity of this space is the same as the reduction of the number of bits used to encode this frame, and we can simply pad this space with empty packets like NULL packets.

This solution is very simple to understand, implement and it preserves the timing synchronization, since we only need to pack the filtered bits of each frame continuously after the picture header and then insert NULL packets until the header of the next frame. However, it inevitably has many drawbacks. First, it can only handle bitrate reduction operations. We only try to fix the header of each frame to its original position, which means the changed frame should not occupy a space larger than the distance between the current frame header and the next. This property does not always hold, since some filtering operations like information embedding and watermarking may increase the bitrate. Secondly, the saved bits are padded with NULL packets to maintain the original constant bit rate and the starting point of each frame, and this ironically runs counter to our initial goal of bitrate reduction. The resulting stream contains the same number of packets as the original one. The only difference is that the number of bits representing each frame has been shrunk, yet this saving is spent immediately by padding NULL packets at the end of each frame.

Here we want to mention that there does exist another approach to bypass the second problem. Up to now we have been using a filter model that is transparent to the client player, which confines us strictly to the MPEG2 standard syntax. However, if some of the filtering intelligence is exported to the end hosts, then some saving can be expected. For example, instead of inserting NULL packets, we may compress them by insert only a special packet saying the next N packets should be NULL packets, and on seeing this packet, a small stub at the end host (right before the client player) takes the responsibility of replacing this packet with the supposed amount of padding packets. (Note that this padding is important to maintain correct timing, especially if the client is using some standard hardware decoding board.) This way, the bandwidth is indeed saved, but at the price of relying on extra non-standard protocol. Of course, this will introduce all problems associated with non-standardized solutions, such as difficulty in software maintenance and upgrading. Therefore, we only consider this as a second choice, and not as a feasible solution.

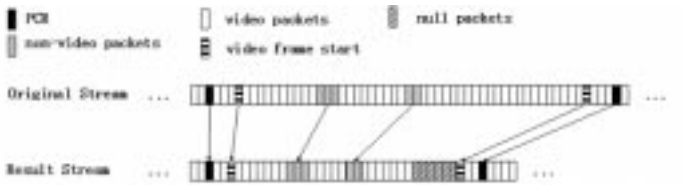

Figure 3: Example: 2/3 shrinking

3.2 Enhanced Solution: Time-Invariant Bitrate Scaling

To ultimately solve the synchronization problem, a more complex algorithm has been designed. The key insight behind it is that we can change the bitrate to another *constant* value while preserving the PCR timestamps by changing the number of packets between any PCR pair to another con-

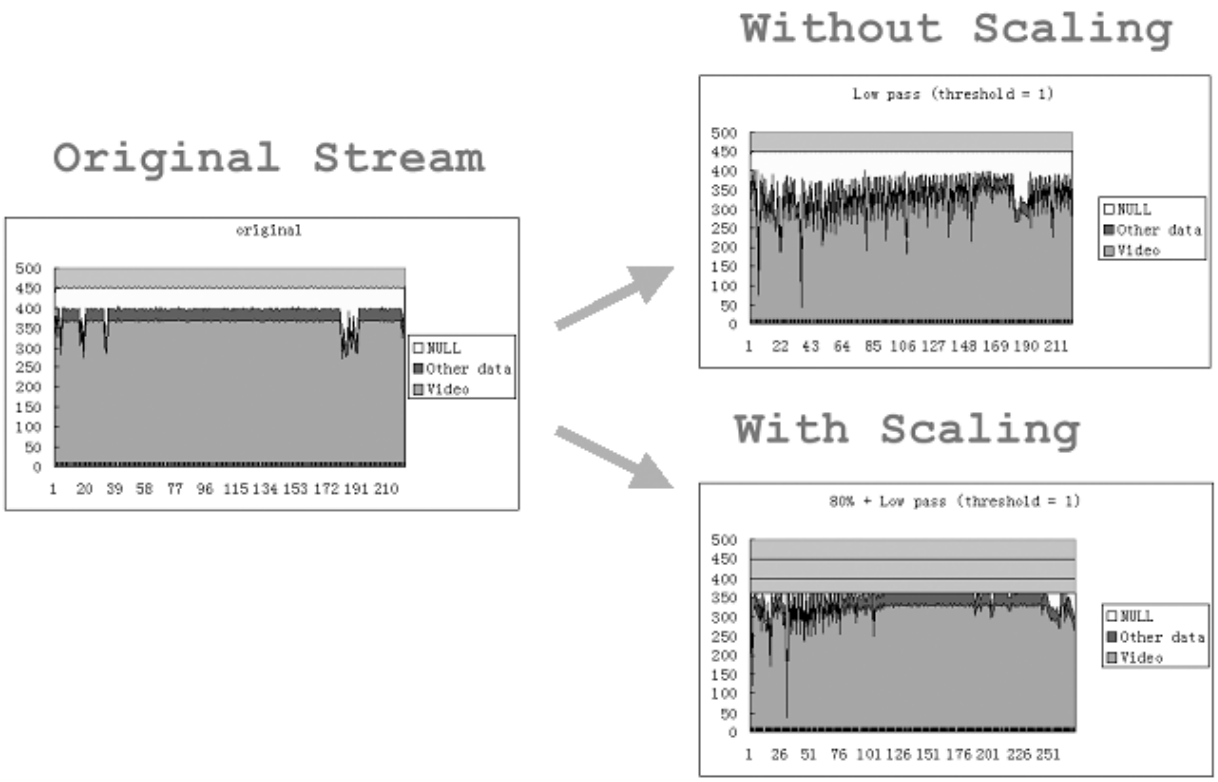

Figure 4: Result of timeline scaling

stant number. This way, we can scale the PCR packets distance and achieve a constant yet different bit rate, as if the time line is scaled looser or tighter to carry more or less packets. All non-video stream packets can be simply mapped to the scaled position that corresponds to the same time point on the scaled timeline as on the original timeline. In case that no exact mapping is available, we could simply use the nearest timepoint on the new timeline without introducing any serious problem. For video stream, the same kind of picture header fixing and frame data packing are conducted as in the first solution but in a scaled way.

An example of shrinking the stream to its 2/3 bandwidth is given in Figure 3. All non-video packets and video packets that contain picture headers are mapped to their corresponding position on the new timeline, and so their distance is also cut to 2/3 of the original. The video packets are filtered and packed closely and as early as possible within the new stream following the header. Intuitively, the filtered video data is squeezed into the remaining space between all non-video packets and picture header packets.

This algorithm is also very simple to implement. For each non-video packet, its distance (in number of packets) from the last PCR packet is multiplied by a scaling factor s, and the result is used to set the distance between this packet and the last PCR packet in the output stream. For video frames, the header containing DTS and PTS is scaled and positioned in the same way, and the remaining bits are closely appended to the header in the result stream.

Now the only problem is how to determine s. If we shrink the timeline too much and for some frames the bitrate reducing operation does not have a significant effect, then again we will not have enough space to squeeze in this frame, which will push the beginning bit of the next frame behind on the time line. On the other hand, if we shrink the timeline too little or expand ($s > 1$) it too much, then more space will be padded using NULL packets to preserve important timepoints, leading to a waste of bandwidth. There exists

one optimal scale factor s_{opt} that can balance these two strengths if it fulfils the condition that

- the filtered frame data will always be squeezed into the scaled stream;
- the number of NULL packets for padding purpose is minimum.

However, this optimal scale factor is hard to estimate in advance since for different operations and different video clips of different scenes, the effect of the filtering on the bitrate can greatly vary.

Therefore, in our current implementation, we simply use a slightly exaggerated scale factor based on the operation type and parameters. For example, for low pass filtering with a threshold of 5, a scaling factor of 0.9 will work almost for all streams. Even if we meet a frame that still occupies more than 0.9 number of packets after the filtering, only the next few frames may be slightly affected. Since a smaller-than-average frame is expected to follow shortly, this local skew can be absorbed by the decoder easily and does not have any chain effect.

Our next step will be looking into how to "learn" this optimal scale factor by analyzing history bitrate change of a stream and adjust this factor s on the fly. It is still not clear how a decoder, especially hardware decoding board, would react if the incoming stream changes from one constant bitrate to another, and it is also an open question how quickly it would adapt to the new rate.

4. EXPERIMENTAL RESULT

Figure 4 shows the effect of the timeline scaling approach. Each point on the x axis represents an occurrence of a PCR packet, and the y axis shows in 3 colors how many video packets, NULL packets or packets for other data stream are in between each 2 PCR packets. We can see that the distribution of the 3 areas is kept almost constant for the original frame except for more NULL packets at the end of a frame. However, without scaling, the number of video packets varies across different PCR intervals and a lot of extra space is padded with NULL packets as shown in the upper right sub-figure. On the other hand, if we do scaling with a scaling factor of 80%, then the padding occurs mostly only at the end of frames and the stream contains mostly only useful data.

One thing we need to point out here is that the skew of access units along the timeline still exist with this scaling approach. What happens is that after the filtering operation, each frame shrinks to a size approximately 80% of its original size. If we mask out all other packets, we can see that in the video stream, frames are packed closely one after another. If one frame takes more space than its share, then the next frame may be pushed behind its timepoint, yet this skew will be compensated later by another frame with a larger shrink effect. As we said before, this kind of small jitter around the exact timepoint on the scaled timeline is acceptable, and it is the change in the bitrate at which the decoder reads in the data that fundamentally makes our scaling algorithm able to solve the problem.

5. CONCLUSION AND FUTURE WORK

In this paper, we have presented our experience in implementing a realtime software filter system for MPEG2 Transport stream. The synchronization problem is introduced and our solutions and experimental results are given.

Our work is a first effort in promoting realtime software filtering of MPEG2 streams, and can be beneficial to many realtime applications that work with MPEG2 system streams like HDTV broadcast.

However, many problems remain to be solved, such as realtime adaptation of the scale factor s. They will be further studied in our future work.

6. ACKNOWLEDGMENT

This work was supported by the NASA grant under contract number NASA NAG 2-1406, National Science Foundation under contract number NSF CCR-9988199 and NSF CCR 0086094, NSF EIA 99-72884 EQ, and NSF EIA 98-70736. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or NASA.

7. REFERENCES

[1] ProxiNet. *http://www.proxinet.com*

[2] J. Smith, R. Mohan, and C. Li "Scalable multimedia delivery for pervasive computing", *ACM Multime-dia*, 1999.

[3] A. Fox, S.D. Gribble, Y. Chawathe, and E.A. Brewer "Adapting to network and client variation using active proxies: lessons and perspectives", *IEEE Personal Communication, Vol. 5, No. 4, pp. 10C19*, August 1998.

[4] W. Y. Ma, B. Shen and J. Brassil "Content Services Network: The Architecture and Protocols", *Proceedings of the Sixth International Workshop on Web Caching and Content Distribution*, June 2001

[5] Nicholas J Yeadon, PhD Thesis. Lancaster University, Lancaster, May, 1996. "Quality of Service Filters for Multimedia Communications", *http://www.comp.lancs.ac.uk/computing/users/njy/thesis/.*

[6] Bin Yu, Klara Nahrstedt. "Realtime Information Embedding of HDTV Stream", *To be submitted to ICME2002.*

[7] ISO/IEC International Standard 13818-1/13818-2, "Generic coding of moving pictures and associated audio information", November 1994

[8] Introduction to HDTV, *http://www.ee.washington.edu /conselec/CE/kuhn/hdtv/95x5.htm*

[9] HDTV-over-IP has arrived! *http://www.2netfx.com/.*

[10] C. E. Holborow "Simulation of Phase-Locked Loop for processing jittered PCRs," *ISO/IEC JTC1/SC29/WG11*, MPEG94/071, March 1994