Online MPEG-2 Transport Stream Multiplexer for .NET

Abstract — As digital television tends to replace the analog one and with continuous development of new and more automated programming technologies, implementation of digital television processing using these techniques becomes simpler and promises to reduce the time and cost of developing such expensive products. This paper presents the challenges imposed by implementing a real-time MPEG-2 multiplexer with .NET using queues and heap data structures and the performance results obtained with the implementation in a high-level language as C#. A fast and flexible algorithm was adapted to construct a constant bit rate multiplexed stream, using multiple files as program sources. Tests developed on two 700 MHz and 1.6 GHz Pentium processors showed that the implemented multiplexer is able to cope with three and, for the second machine, five real time program streams before processor overload occurred. The portable .NET implementation allowed for testing on different systems. Being rather an unusual approach, the C# implementation tries to prove that such an approach is not only possible, but also viable for small, low-cost real time systems.

I. PROBLEM STATEMENT

The widespread of digital television equipments and the almost global availability of satellite DVB broadcasting opens the way for developing digital television processing software on common inexpensive PC hardware. Digital television signal processing is generally performed on expensive tenths thousand dollar priced equipment. The article shows that real time multiplexing of already encoded streams can be performed with rather inexpensive hardware and using a high level programming language as C# for the .NET platform.

An example model can be that of a local area network (figure 1), in which there is one computer that encapsulates, in real time, many streams, and delivers the high bit rate multiplexed stream to a local area network. There are one or more clients that filter the desired program and watch it on line (classic digital television). This is the model used, for example, in VideoLAN ([1]), an open source video streaming solution for local networks. However, VideoLAN is not able to multiplex more programs and then deliver them locally. It can transcode and stream one single Transport Stream, or it can filter out programs it receives from a DVB Transport Stream. The latter case is not multiplexing.

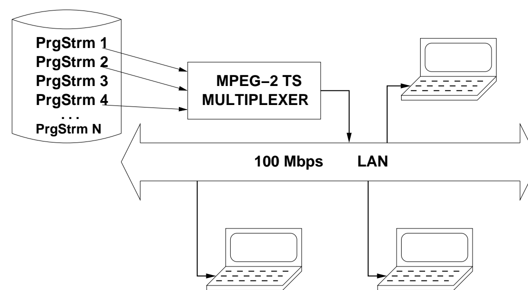

Figure 1: Broadcast architecture.

To ensure DVB conformance ([2]), the input streams are in fact program streams (PSs) composed of only one video and many audio streams. The generated multiplexed stream is a transport stream (TS) which comprises many elementary streams (ESs). As the standard requests ([3]), the audio and video ESs are further encapsulated into packetized elementary streams (PESs). The packetization not only produces limited-size blocks of data but attaches a time stamp to each PES packet beginning with the start of an access

unit (which is either a picture, when referred to video, or the set of samples that describes an audio PCM sequence). The PS is composed of video and audio PESs with consecutive time stamps (indicating the exact timing of the PES packets).

The output multiplexed TS is composed of successive packets of fixed size, typically 188 bytes, ordered by their global time in the target stream. One possible representation of a particular creation process is given in figure 2.

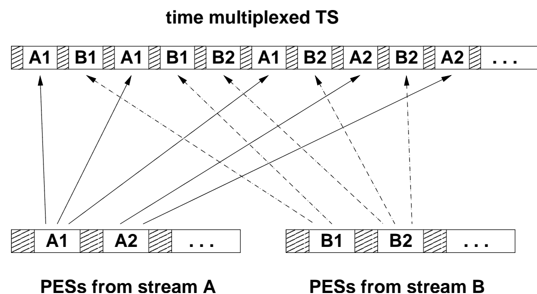

Figure 2: Transport Stream structure: one TS packet contains a header and a payload that embeds one part of a PES packet.

Input PSs usually have different, variable, bit rates. In the multiplex stream, PES packets of one program should keep their initial order. The bytes from that program should be delivered at their original (possibly variable) bit rate. If there are n sources, assume t_{src_1}, t_{src_2}, …, t_{src_n} are their current times. When constructing the multiplexed target stream, assume t_{target} is the multiplexer's current time. There are three situations:

- there is only one source i for which:

$$t_{src_i} \leq t_{target} < t_{src_k} \quad \text{where} \qquad (1)$$

$$k = 1 \dots n, \quad k \neq i \qquad (2)$$

  The PES packet from the source i with the smallest time is thus chosen to be put in the target transport stream.

- for all sources, the following relation stands:

$$t_{target} < t_{src_k}, \quad k = 1 \dots n \qquad (3)$$

  All the sources have packets whose time has not arrived yet. One empty packet is put in the target transport stream and global target clock is advanced.

- we have p sources, $p \geq 2$, $i_1 \dots i_p$ for which:

$$t_{src_i_j} \leq t_{target} < t_{src_k} \quad \text{where} \qquad (4)$$

$$k = 1 \dots n, \quad k \neq i_j, \quad j = 1 \dots p \qquad (5)$$

  In this situation, for a short period of time, the sum of all source bit rates exceeds the given global target bit rate (which is assumed to be constant - CBR). In practice this situation should never occur because this could cause certain packets to be inserted in target stream later than required. De-synchronization problems and interrupts in the audio / video streams may occur. For this situation statistical multiplexing is used, video or audio bit rate is reduced by imposing quality constraints on the video encoder ([4]).

An audio sequence is composed of successive compressed frames, PESs, each bearing a specific time stamp indicating its intended presentation time ([3], [5]). A time stamp in the audio sequence is greater than all previous ones. For a video sequence, depending on the coding pattern, each frame can be:

I intra coded, encoded independently of other frames;

P predicted, encoded as a difference from previous frame;

B bidirectionally predicted, encoded as a difference from both previous and next frame.

This means that the display order of the compressed frames differs from the decoding order of the same frames. Usually every frame begins in a new PES packet. So for a PES packet we can have presentation time stamp (PTS), which is the intended time of display of the frame beginning in the PES packet, and decoding time stamp (DTS), which indicates the preferred decoding time for the frame beginning in this PES packet. Thus, an audio or video PES packet can be classified regarding his PTS and/or DTS:

- **a PES packet has both PTS and DTS**, indicating this PES payload begins with an I or P picture start code. The frame is used as a reference in decoding other frames so it comes "out of order". Its decoding time is earlier than its presentation time. The DTS, not the PTS time stamp, indicates the time "order" in PES stream;

- **only PTS**, for a video B picture or audio frame that start in this frame. The decoding time is the same as its presentation time;

- **none**, for this frame the time stamp will be computed considering closest time stamps from packets above and behind him.

The PES time stamps (PTS and DTS) within program streams (PSs) are given as a multiple of 1 / 90 kHz. However, transport stream (TS) program clock reference (PCR) timestamps reflect multiple of 1 / 27 MHz, being much more accurate than PTS and DTS. The TS embeds more than one program; each of its programs itself embeds a program clock reference (PCR) time stamp in its TS packets from time to time. PTS and DTS from a specific program will be synchronized to the PCR of that respective program within the TS.

The TS will be constructed from one or several programs, each program containing valid PTS and DTS timestamps. For that, each PES packet (usually having sizes from 1KB to 64KB and even higher) have to be split into several hundred TS packets (given that a TS packet size is 188 B)

PES streams can be of constant (CBR) or variable bit rate (VBR); given two consecutive PES packets A and C, the first byte of PES A is byte a in stream and the first byte of PES C is byte c in stream. Their timestamps will obey the relation $timestamp_a < timestamp_c$. For any byte b in stream in between byte a and c, its intended time to enter a hypothetical decoder is (as described by [3]):

$$t_b = \frac{timestamp_a}{90\text{kHz}} + \frac{position(b) - position(a)}{rate_b} \tag{6}$$

where $position(a)$ is the byte position in stream for byte a, the same for bytes b and c, and $rate_b$ is given by:

$$rate_b = \frac{[position(c) - position(a)] \times 90\text{kHz}}{timestamp_c - timestamp_a} \tag{7}$$

Also as described by [3], a TS program byte e which is between two bytes d and f, as first bytes of packets characterized by PCR_d and PCR_f, the intended time t_e of entering the system target decoder is ([3]):

$$t_e = \frac{PCR_d}{27\text{MHz}} + \frac{position(e) - position(d)}{rate_e} \tag{8}$$

, where $position(d)$ is the byte position in transport stream for byte d, the same for bytes e and f, and $rate_e$ is given by:

$$rate_e = \frac{[position(f) - position(d)] \times 27\text{MHz}}{PCR_f - PCR_d} \tag{9}$$

In this paper we present a simple but fast approach to generate multiple program transport streams (MPTS) using queue buffers for packets and a min-heap for deciding the next packet to be inserted in the TS. It is a variation of the algorithm whose principle is presented in [4] and some other variants implemented by [1] and [6]. Reference [4] deals with statistical multiplexing; this does not apply here since the transport stream is constructed from Program Stream files, already encoded. We use that to avoid repacketization of PESs. The VideoLAN project ([1]) does have a multiplexer, but for a single program only. We extend the principle implemented there. As for [6], it is an industrial implementation optimized to give the best performance. We will refer to it later.

This paper's objective is to present a concise algorithm taking advantage of the pre-packetized structure of source streams and the performance achieved using a high level language. This paper is organized as follows: section 2 describes the simpler algorithm used to generate a single program transport stream (SPTS), section 3 exposes the mechanism for creating the multiple program transport stream (MPTS) using queues and a min-heap, section 4 presents the implementation requirements and experimental results; eventually section 5 shows the advantages and disadvantages of our approach and some future work.

II. SPTS MULTIPLEXING ALGORITHM

The simplest approach in generating a TS is to use only one PS as source. The resulted stream, single program transport stream (SPTS) will have only one program and of course one associated PCR. For each packetized elementary stream within the PS we associated a queue buffer. Each of these buffers also has an associated *expiration time*, which is the time when first data bytes must leave the queue. The queue buffer has one or many PES packets, each with its associate time stamp. This way, an expiration time may be attached for each byte in queue. This is the time of entering the standard target decoder.

Generation of SPTS is not limited only to pooling queues and choosing the right one. The SPTS contain also PCR time stamps from time to time (at least ten times a second) and program association table (PAT) and program map table (PMT) which describe the content of the TS ([2]). When none of the queues contain data that expire and there is no need to put a PAT or a PMT, a NULL packet is put into stream. We would not detail these aspects here. More details on that can be found in [3] and [7]. The general algorithm is presented in figure 3. All of the details of the implementation were not included, to keep the exposure concise.

$clock \leftarrow 0$
while not $empty(source)$ **do**
    while none of the queues is full **do**
        read PES packet
        check packet type (audio, video, ..)
        put packet to corresponding queue
    while none of the queues is empty **do**
        find x where $queue_x$ expires first
        if $time_x \leq clock$ **then**
            insert packet from $queue_x$ into TS
        else if PAT period elapsed **then**
            insert PAT into TS
        else if PMT period elapsed **then**
            insert PMT into TS
        else
            insert NULL packet into TS
        $clock \leftarrow clock + \frac{TS_packet_size}{TS_bitrate}$

Figure 3: SPTS multiplexing.

The main drawback of this algorithm is that for each TS packet put in the transport stream, a minimum of all queue expiration times is computed. This minimum can be calculated and made available before each step.

III. MPTS MULTIPLEX ALGORITHM

The creation of a MPTS assumes there is more than one program, each of which will have a PCR of its own. Every program will start at a certain initial time stamp and then increase, as long as the stream flows, with the same amount per packet (since a CBR MPTS is created). The initial PCR time stamps for each program are created using the time reference of every source ([3, 8, 9]).

A typical multiplexing process is presented in figure 4. As opposed to the SPTS case, where the data flow was built in a single process, the MPTS makes use of $N+1$-th processes, where N is the number of PS sources. N processes are delivering data to their corresponding queue buffers, keeping them as full as possible. The $N+1$ process is the multiplexer itself. When a source process cannot write the current PES packet from file to the specific queue since this later one is full, it is put to sleep. It will be awakened when the multiplexing process reads at least one slot from this full queue buffer.

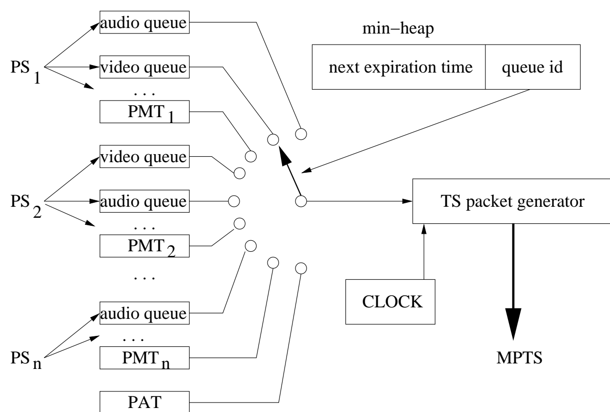

Figure 4: TS multiplexing process. Source decision is taken using a min-heap criteria.

The multiplexer process will be aware of expiration time for all queues. These times are kept in a min-heap structure, making the extraction of a minimum as fast as $O(1)$. Also, when the heap is updated, its operation is kept to $O(log\, PIDs)$, where $PIDs$ is the heap dimension, that is the number of elementary streams considered. In fact, the min-heap stores pairs in the form of {expiration time, queue id}, since the identifier of the minimum expiration time queue is needed to extract payload from that queue.

The multiplexer process simplified description is presented in figure 5.

Two things must be considered about the algorithm (5):

- the min-heap property is maintained not comparing PES $timestamp_x$ for all program streams, but the difference $(timestamp_x - initial_timestamp_x)$ of each stream, thus es-

```
for all sources x
    clock_x ← initial_time_for_source_x
while there is at least one nonempty queue do
    inspect min heap's root {time_x, queue_x}
    if time_x ≤ clock_x then
        insert packet from queue_x into TS
        recompute latest_time for queue x
        remove root from min-heap
        insert {latest_time, queue_x} into min-heap
    else if PAT period elapsed then
        insert PAT into TS
    else if one of PMTs period elapsed then
        insert that PMT into TS
    else
        insert NULL packet into TS
    for all sources x
        clock_x ← clock_x + TS_packet_size / TS_bitrate
```

Figure 5: MPTS multiplexing process. Each program has a different initial time.

tablishing a common time base for min-heap criteria;

- not a whole PES packet is inserted into the TS at a time. Since the PES packet is several hundred times longer than a TS packet, only a part of the PES is put to the TS. The multiplexer will watch that every PES starts at the beginning of a new TS packet.

These details and other implementation issues were intentionally left out of the presentation of the algorithm from figure 5, to maintain the clarity of the presentation.

IV. EXPERIMENTAL RESULTS

The main objective was to maintain multiplex speed as closest as possible to the real bit rate of the target stream. Experimental results showed that this speed surpasses the real bit rate of the transport stream in certain cases.

The implementation throughput whose sources are not live streams but files is given in table 1. Usual throughput of a satellite transponder is 38 Mbps. In this case, all processes would be limited by the imposed transmission bit rate (38 Mbps, for example).

Nr. of streams tested	Case	Total size [MB]	Mux speed [Mbps]	Mux time [s]
1 (A)	exp.	78.5	233.8	2.81
	ref.	75.3	451.1	1.4
2 (A, B)	exp.	195	115.6	14.15
	ref.	187	295.8	5.3
3 (A, B, C)	exp.	233.2	67.14	29.14
	ref.	212.7	110.8	16.1

Table 1: Experimental performance versus reference performance on a Pentium IV 1.6 GHz. Three streams were used and experimental performance was compared with Manzanita Software as reference. Target MPTS bit rate was 38 Mbps in both cases.

Three PS files were chosen, 150s each, with bit rates of $rate_A$ = 4,3 Mbps, $rate_B$ = 6,5 Mbps and $rate_C$ = 2,1 Mbps. On a Intel Pentium IV 1.6 GHz machine, with 512 MB RAM, experiments showed the results presented in table 1. The results show that the reference implementation [6] performs about twice as faster as the experimental implementation. We consider this an acceptable result since it is a .NET implementation.

Target bit rate was maintained constant, 38 Mbps for all three cases. As total size of the generated bit stream increases, the multiplex speed drops, but it is higher than the target of 38 Mbps.

Tests were conducted on an extreme configuration with programs taken from a satellite transponder with programs of 4-5 Mbps each showed that on a Pentium III 700 MHz machine the processor load was as follows (table 2):

Nr. of sources	Bit rate [Mbps]	Processor load [%]
1	4.8	32
2	9.7	78
3	14.3	100

Table 2: Performance on a Pentium III 700 MHz.

Experiments made showed that for a medium range computer (700 MHz) three programs overload the processor and the multiplexer started loosing synchronization. Even on powerful machines (1.6 GHz) the number of processed streams

does not reach 6-8, which is typical in a 38 Mbps stream, before the processor becomes overloaded. Only 5 streams could be multiplexed at once but the processor load was 98%. Although there is a significant overhead because of the particular C# .NET implementation chosen, an inexpensive machine could be used for simple extraction and then selective recomposition of a transport stream. In all situations, when processor load surpassed 99%, audio and video desynchronization affected all streams.

V. CONCLUSIONS AND FUTURE WORK

This paper describes a straightforward approach for multiplexing several program streams using a queue and a min-heap implementation in .NET. It takes advantage of the already packetized structure of the programs it processes. To test the performance of the algorithm a C# implementation of a real MPEG-2 TS multiplexer was developed. This was able to multiplex five program streams of medium bit rates (3 - 6 Mbps) faster than real time on a Pentium IV 1.6 GHz. The limitation of a low performance Pentium III 700 MHz machine is at maximum three live streams, whereas the more powerful 1.6 GHz machine was able to multiplex five streams. Tests showed that the implementation is not much slower than the reference industrial implementation using all optimizations possible.

In our future work, we will explore the possibility of using the Java language and different Java Virtual Machines to test the multiplex throughput. This could have implications in processing some degree of transport stream processing using Java for the MHP (Multimedia Home Platform). The experiments showed that multiplexing several program streams with a low performance computer is feasible even with high-level language implementations like C#. Future developments will try to detect situations for which bandwidth overflow occurs and to discard the packets of a pre-selected program source.

REFERENCES

[1] Alexis de Lattre et al., *VideoLAN Streaming Howto*, http://www.videolan.org/doc/

[2] ETSI TR 101 154, *Digital Video Broadcasting (DVB); Implementation guidelines for the use of MPEG-2 Systems, Video and Audio in satellite, cable and terrestrial broadcasting applications*, European Broadcast Union, July 2000

[3] ISO/IEC 13818-1, *Information technology - Generic coding of moving pictures and associated audio - Part 1: Systems*, Technical Report, 1995

[4] J. Watkinson, *MPEG-2*, Butterworth Heinemann, 1999

[5] ISO/IEC 11172-3, *Information technology - Generic coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s - Part 3: Audio*, Technical Report, August 1993

[6] Manzanita Software, *MPEG-2 Transport Stream Multiplexer for Multiple Programs*, http://www.manzanitasystems.com/mp2tsmm.html

[7] ITU Radiocommunication Study Groups, *A guide to digital terrestrial television broadcasting in the VHF/UHF bands*, Technical Report, 1996

[8] ISO/IEC 13818-2, *Information technology - Generic coding of moving pictures and associated audio - Part 2: Video*, Technical Report, 1995

[9] R. Schäfer, T. Sikora, "Digital Video Coding Standards and Their Role in Video Communications", *IEEE Proc.*, Vol. 83 No. 6, June 1995