A Pyramidal Scheme of Residue Hypercubes for Adaptive Video Streaming

Adrian Enache, Costin A. Boiangiu

***Abstract*—**The following paper proposes a solution for the adaptive video streaming problem based on a construction of a pyramid of hypercubes. The hypercubes are built as residues between successive downsampling and upsampling operations over chunks of video data. The described mechanism allows a great deal of flexibility with minimum overhead, is both general and mathematically elegant and may be further refined for better transmission control.

***Keywords*—**adaptive video streaming, entropy coding, hypercube residue, realtime video protocols

I. Introduction

THERE are a number of video file formats, encoders and decoders and internet video transmission methods available today, each with advantages and disadvantages. From the end user's point of view, a video streaming service should offer low latency (so they will not hear the neighbors getting excited about a goal and actually seeing it seconds later), high quality (low information loss and few coding artifacts), possibility to move through a video and some extra information contained in the video, like subtitles and chapters.

From the technical point of view, there are more layers which work together in order to offer the entire service [1], [2].

Starting, top to bottom, with the application layer, the outer most box is the video file format or video container, which is a meta-file containing information needed for a system to interpret the actual video (or/and audio) data. Here is where the information about aspect ratio, frame rate, key-frames, and duration is stored, usually in discreet packets, which are useful for streaming, and extra information, like chapters. The most known containers are AVI, MP4, MOV, and FLV. AVI does not offer the functionality to skip over un-downloaded video segments and MOV and MP4 evolved to be the same, along with FLV offering all is needed for video streaming.

This article moves from here lower, offering an alternative to the encoding and decoding mechanism (codec) and proposing a way to use the available protocols in order to offer a stable high-quality low-latency unicast or multicast video streaming service.

Today's high quality video codec standard is H.265 or HEVC (having predecessor the H.264 which is being used in most of the popular video streaming sites, like YouTube) which is a very advanced version of the initial MPEG. The MPEG codec splits all the frames in a video sequence in more types: I (initial, which contains all information in an image) P (predicted, which contains the difference of information between the predicted and actual frame) and B (bi-predicted, which is constructed from the I and P or P and P frames). Each of these frames is split in small 8x8 pixel blocks which are compressed with a discreet cosine transform (like in JPEG). Larger 16x16 pixel macro-blocks are used to estimate movement of objects along 4 blocks, in order to predict more accurately the P and B frames. The HEVC codec uses small block dimensions of variable size (1-64 pixels), uses intra-frame predictions (constructs the new frame considering neighboring colors from inside the frame) and more accurate intra-frame predictions estimating the objects movements better.

The drawbacks of this entire mechanism are complexity and single frame image quality. At the time when surveillance cameras moved from Motion Jpeg to MPEG2, the recording device could not handle the video from all the mounted cameras and dropped frames. In a system where some frames could be important (like a thief running in front of the camera) this was not acceptable. The other larger disadvantage was that the actual frames didn't contain enough useful information. The codec was designed to look good, by predicting, but when a person's face needed to be observed, no single frame could offer the information because of the high compression.

That is why codecs like Motion JPEG (and the more advanced form MxPEG) still have their place. They encode each frame as JPEG images and use temporal coherence in order to save space without losing quality. Thus every frame contains information which could be useful in the case of a split second event, but at the cost of bandwidth and memory.

These are the two major codec directions that are widely used; both have the ability to move through video and to stream, one inherently (MxJPEG, because it contains all the images) and the other with the aid of key-frames, but none can be used with an unreliable network. They can only be sent at the quality they exist on disk, thus a fluctuation of connection speed halts the entire playback.

Getting lower, to the transport layer, there are two major protocols in use: TCP and UDP.

TCP is a connection oriented protocol aimed at being reliable. Which means that every time a packet is lost, corrupted, or comes in the wrong order, it is being resent. This is useful for data transmission, but awful for live streaming, as

the time lost with overhead and retransmission generates halts in the playback. If a connection failure occurs, in order to recreate the TCP connection takes a full 3 seconds.

The other protocol, UDP, is developed having just that in mind: it is better to lose a frame during a stream than to wait for the respective frame to be retransmitted. Thus UDP is very simple, packet oriented (it doesn't have to wait to establish a connection), that just sends the package to the destination without caring if it actually arrives. It has the advantage that it can be used in multicast scenarios, where the server sends only one copy of the file to a group IP address and the routers and switches take care of multiplying the file to all the computers being interested in the file.

With all the advantages mentioned of UDP over TCP in streaming videos, sites like YouTube use protocols over HTTP (like: Adobe Dynamic Streaming or DASH) which are on top of TCP. So it is not as much streaming as downloading the next chunk faster than it is played and selecting from a series of qualities in order not to fall behind.

The proposed system aims to incorporate as much as possible from both approaches, whilst being fine-tuned for streaming at adaptive quality and taking advantage of protocols developed specially for streaming, like RTSP (used for sending commands) with RTP (used for sending the actual video data) and RTCP (used for sending feedback information useful in changing the maximum quality sent by the server).

At the IP level, this approach can be used to send different quality video streams to the users in the same group of a multicast address, as each user takes only as much as it can.

II. Problem Formulation

Thus the problem we are addressing is finding a method of optimally transmitting video streaming data to the client such that we only pass a minimum amount of data, while also taking into consideration fluctuating channel bandwidth limitations. In this sense, the current paper presents a different method based on sending chunks of frame data to the streaming client. This is done by gathering a series of frames from the video input and processing them, the result being then transmitted incrementally as different level of detail structures [3].

The proposed method also presents an implicit way of handling adaptiveness to the transmitting channel bandwidth. This is due to the fact that the main algorithm tries to split high frequency details on different levels. These details are sent in an importance-based fashion resulting in the fact that, even if not all the data gets transmitted successfully, the user can still get a less detailed view of the video data. This idea is loosely based on [4] where single image's color components are compressed by downscaling and restoring the original image scale with a predicted residue from other color components. It also expands a paragraph that describes video transmission by splitting in resolutions from [5].

III. Problem Solution

We start by taking a known video format and extracting all the uncompressed frames from it. We number these frames from 1 to N. We will divide this interval $[1, N]$ of frames into smaller, variable size subintervals that will be used as input for our algorithm.

We first define the dimensions of the matrices (named here frame cubes) that we are going to construct using those subintervals. As seen in Fig. 1, we create the 3-dimensional matrix featuring the frames from a whole subinterval. This means that a frame cube has a width W and height H, both inherited from the starting frame format. These 2 dimensional sizes will remain constant because the format of the input does not change during the playback. The 3rd dimensional size of the cube is duration D, expressed as the length of the associated subinterval. As such, we will have the input data represented as a series of frame cubes that are then processed by the server and transmitted to streaming clients.

We use the following notation to represent the frame cubes:

- $(W, H, [1, C_1])$ starting on frame 1, ending on frame $C_1, D = C_1$
- $(W, H, [C_1 + 1, C_2])$ with $D = C_2 - C_1$
- ...
- $(W, H, [C_n, N])$ with $D = N - C_n + 1$

A. Algorithm

The main processing algorithm on the server consists of applying multiple iterations to the starting frame cube. One such iteration (seen in Fig. 1) contains the following steps:

1. Apply a downscale operation on the input frame cube A, resulting in another smaller frame cube B
2. Apply an upscale operation on frame cube B. This operation creates frame cube C that has the same dimensions as frame cube A, but differs slightly in data that was lost during the downscale operation
3. Compute a residue frame cube as the difference of frame cubes R = C – A

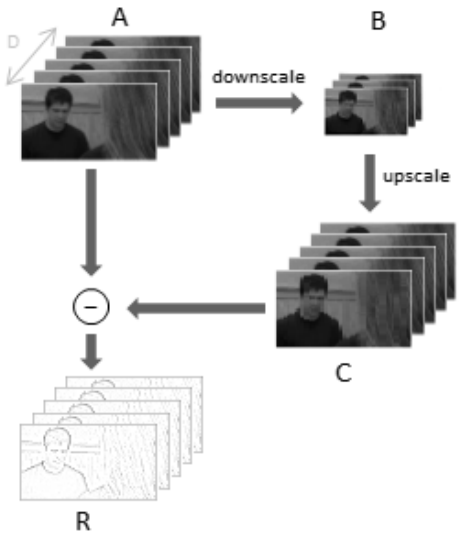

Fig. 1 Hypercube generation

Using this process, we continue applying it to the smaller frame cube B that resulted from the previous iteration, creating each time a new residue cube R. The iteration stops either after a given number of steps have occurred, or when the dimensions of the resulting frame cubes are lower than a set threshold.

Consider that we take a frame cube $(W, H, [C_i + 1, C_{i+1}])$ with $D = C_{i+1} - C_i$, and apply the algorithm with a scaling factor of 2. This means that we get a residue frame cube series with dimensions $(W, H, D), (W/2, H/2, D/2), ..., (W/2^n, H/2^n, D/2^n)$ and a base frame cube having the dimensions equal to half the smallest residue cube $(W/2^{n+1}, H/2^{n+1}, D/2^{n+1})$. We mark these residue frame cubes R_0, R_1 … R_n based on the power of the scaling factor that was applied.

Having this cube represented as a hierarchy of residue frame cubes (named residue hypercube) and a base cube, we can now transmit to the client the base cube and a number of residue cubes (hypercube slices) starting from highest index R_n to R_0. The client will use this data to synthesize the video frames. The reason why we transmit residues in this order is because of possible bandwidth limitations. If there were insufficient bandwidth, the client would still receive enough information to reconstruct the original frames at a lower quality (even to the level of getting and rendering just the base cube).

In order to reduce the size of the transmitted data, on the server side, we can compress both the residue frame hypercube and the base frame cube beforehand. The compression ratio of the base cube is small because of its small size and complex information; however, the residues can be compressed to a high degree because they contain small values (that actually encode the small details lost in the downscale operation). As such, various methods of entropy encoding can be used successfully in order to compress the residues. One such algorithm that works well is Huffman coding. Ideally, a lossless compression algorithm should be used for the residue hypercubes because information itself might be lost due to the bandwidth limitations.

On the client side, we need to use the received residue hypercube and base cube to synthesize the original frames as best as we can. As seen in Fig. 2, we mainly apply the inverse steps used on the server side.

This means that we need to decode/decompress the residue hypercube and then, starting with the base cube B, upscale to A_0:

1. Add the corresponding residue frame cube in order to get a frame cube of the current resolution.
2. Apply an upscale operation resulting in cube A_i
3. Iterate first 2 steps in order to consume all the received residue hypercube slices

This synthesizing algorithm will result in a frame cube of size $(W/2^i, H/2^i, D/2^i)$ where i is determined by the number of residue hypercube slices that have reached the client in a useful timeframe. This timeframe can be expressed as the difference between the current time and the time when the rendering of the last frame cube had finished. The client can also artificially cut the reconstruction process if the timeframe ends. If this did not happen, the client renderer would show stuttering or pausing for period of time until it can render new frames.

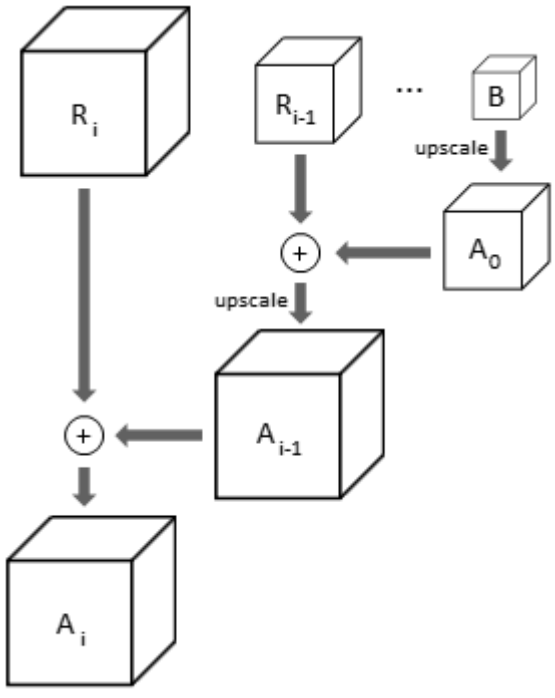

Fig. 2 Codec Workings

After the synthesizing of the current cube has finished, the client can render the frame cube (at the original frame rate, doing any required interpolation). In this sense, the receiving

and synthesizing of data happens in an asynchronous fashion with respect to the actual frame rendering.

B. Implementation Details

There are a number of issues that arise when using the proposed method.

First of all, considering that we have a dynamic range $[0, V_{\max}]$ in the original frame data, the residue data will have a dynamic range of $[-V_{\max}, V_{\max}]$. This effectively doubles the range, which means that we need double the space to store and transmit the residue hypercubes. In practice though, this problem is resolved by the compression algorithm. Even though we have a bigger value range, most of the residue data values are small, because the interpolated frame resembles very well the actual large frame.

However, we propose two mapping functions that can be used in order to create a lossy version of the residue data.

Both of the functions are defined as $f : [0,512) \to [0,256)$ and can be applied on input values that were translated by $V_{\max}$ (which in our case is 256 because we are working with RGB 8bit values). In other words, the residue values which are distributed mainly around 0 will be redistributed around 256. This means that we can apply an importance sampling schema which takes in consideration the prior knowledge of the distribution.

The first function that gave good results is based on a Gaussian distribution with the highest density of values near 256. The function is presented in Fig. 3 and has the following form:

$$f(x) = 256e^{-1.5\pi q^2}, q = \frac{x-256}{256} \quad (1)$$

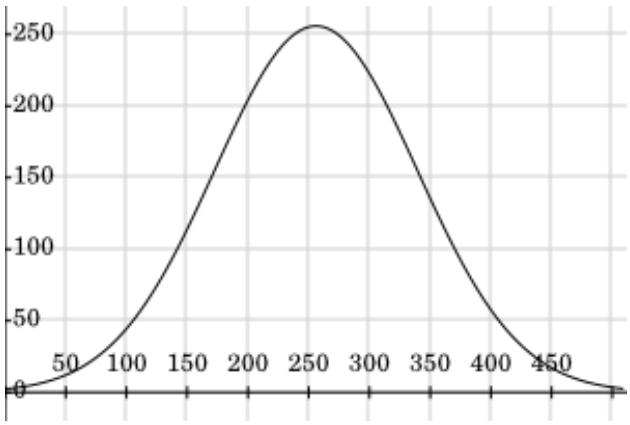

Fig. 3 Gaussian Distribution

The second function is derived from the sigmoid function. It is presented in Fig. 4 and has the following form:

$$f(x) = 256 - 512\left|\frac{1}{1+e^{-kq}} - 0.5\right|, q = \frac{x-256}{256} \quad (2)$$

In addition we apply a first order derivative constraint on the x=256 value. The constraint is:

$$\left.\frac{df(x)}{dx}\right|_{x=256+} = 1 \quad (3)$$

This means that the mapping for values close to 256 will be one to one. This way the function keeps exact precision near the midpoint 256 value (which is the original high 0 value distribution) but loses more precision near the ends of the interval, which are less important. This tends to perform better than the Gaussian variant because of the "exactness" near the peak distribution point.

Applying the derivative constraint, we get $k = 4$ and the final function form is:

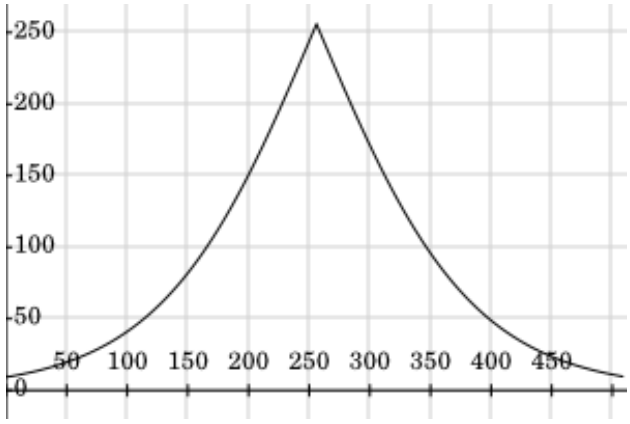

Fig. 4 Distribution that keeps the 256 point intact

The second issue that arises is that the residue values are close, but not always equal to 0. For this problem we can again apply a lossy transformation, such as thresholding low values (e.g. anything lower than 5 can be considered a 0). This transformation can be applied on top of the remapping that we just discussed in order to get even higher compression rates (because of the lower entropy).

Since this destroys data, it will generate possible artifacts at the synthesizing operation. In order to minimize this, we can, instead of replacing the thresholded values with 0, replace them with some special values. This operation would also lower data entropy, but it has the advantage that we know which values we have thresholded. Using this information, we can apply local methods to reconstruct the destroyed values. These methods should only be applied on all but the last residue addition operation because they would otherwise introduce other visible artifacts in the final frame cube.

The last issue that appeared after running the demo implementation was that, in the case where the client does not receive the complete residue hypercube, hard scene cuts

become linearly interpolated. This may not be an important problem in itself, but we can limit the effect by setting subinterval cut points on the same frames as hard scene cuts. For hard scene cut detection a method based on Otsu maximal entropy can be used with good results; or for precision, we can rely on the already created pyramid to use pyramidal correlation [6].

IV. Results

For the implementation, the OpenCV C# wrapper "Emgu CV" was used for the frame extraction and manipulation. The technique that the demo uses is offline because the residue hypercube construction is saved to a file prior to transmitting slices from it.

The input video subinterval had 149 frames in RGB 8bit format (640x360 resolution), that in jpeg form consumed 7.23MB of data. The algorithm hence used 32bit integers as a base data type for the cube cells. The base cube has 73 frames (hence dimensions 320x180x73), built after one iteration, taking 1.45MB in jpeg format. The level 1 residue cube (with 2 bytes per channel, compressed with gzip; dimensions 640x360x149) takes 3MB. Thus the transmitted network size is 4.45MB. This gives us a compression ratio of 1.62 to 1 in this case.

V. Future Improvements

A possible improvement would be to reduce the granularity of the hypercube construction in terms of input frames. Such a method would build different cube depths for different regions of the input frame subinterval. E.g. if we detect regions with less entropy, i.e. smaller number of colors in the same region, a hypercube constructed for that region would need a lower number of slices because there is not much data to be lost when downsampling.

Another possible improvement can be based on changing the color space from RGB to YCbCr and encoding the chrominance channels with half the dynamic range, like in the common jpeg images. The human eye is less sensitive to chrominance, which means that the double range problem is no longer present for Cb and Cr. The luminance channel would still have double the dynamic range in residues so we can choose to remap that with one of the provided functions. Also, the hypercube itself can be compressed further in the manner presented here [7] with better than jpeg2000[8] quality and the local window size can be automatically set by interpreting the standard deviation graph given by increasing windows around a pixel in 3D space instead of 2D paper [9].

From the user experience point of view, methods other than scaling can be used to generate better quality residues [10] or we could try to preserve sharp lines by filtering in a different domain [11]. Reference [12] discuses just the 3D data case methods we need in order to display as much quality as possible from less residues.

Finally, from a purely implementation-bound perspective, a major improvement that may change the server behavior from offline encoding to online would be to use GPGPU techniques [13] in order to construct the residue hypercubes. This would mean that all the resizing is done on the GPU and, considering that most of the new graphical boards support 3D textures, this operation can be completely offloaded. The CPU will just need to run the online encoding scheme (in both the server and client cases).

A. Network Transmission

The base layer can be transmitted using a TCP connection, as it contains very little information and the probability of not being able to send the video at that quality is very low. This offers a stable very low quality video feed, without introducing transmission errors, which at this phase can ruin the entire cube.

The residual layers are sent through RTP over UDP. The Realtime Transmission Protocol offers ways to restore the packet sequence through the 16bit SequenceNumber field and also to synchronize with the lower resolution layers through the 32bit Timestamp field. Lost resolutions are approximated with same resolutions from a past hypercube, if the scene didn't change; are dropped if it is a high resolution; or are being retransmitted in the rare case of scene change and low resolution loss and only if the connection speed can handle the retransmission without stopping the video.

In Fig. 5 is an example where the upper left half of the image lost the 4-th resolution layer. The change in quality is dimmed by a small blur that is hardly visible.

Fig. 5 Bottom-right half lost 4-th resolution layer (original image taken from "*Wikimedia Commons*")

On the client machine, based on the Time stamp and Sequence Number, a high quality packet can be dropped, if the player must display the following frames.

If a single user is receiving the video, RTCP is used to analyze the quality of the connection, and in the case that the client frequently drops high frequency cubes, the server stops transmitting them, saving bandwidth and CPU power on the server side also.

In the case of multicast, each user takes as much as it can handle.

To complete the Realtime Transmission Protocols chains, RTSP can be used in order to offer the user ways to navigate through the video. Each hypercube can be viewed as a key-frame, offering high granularity traversal. The low quality video being of small size can be feed to the user after just a few seconds of downloading.

VI. Conclusion

The proposed method of video streaming leverages the advantages of sending blocks of frames at different resolutions using a differential schema of encoding. The residue hypercubes are constructed using a number of complete frames and the slices are transmitted in order of importance over the channel. A network transmission method using TCP for low quality stable video, RTP over UDP for sending details, RTCP for quality logging for saving server power and RTSP for navigating is also provided.

The coding and decoding are very non-CPU intensive and preserve quality of individual frames when transmitted at the top quality and when losing a residual layer, the quality loss is small. The video is quality adaptive at the client directly and in case of a very poor connection, it can save server resources also.

References

[1] H.-J. Yang, H.-C. Lin, Y.-D. Wang, L.-H. Kuo, "Designing and constructing live streaming system for broadcast," in *Proc. 2010 American conference on Applied Mathematics* (AMERICAN-MATH'10), Harvard University, Cambridge, USA January 27-29, 2010, WSEAS Press, pp. 266-271.

[2] H. H. Soliman, H. M. El-Bakry, M. Reda, "Real-time transmission of video streaming over computer networks," in *Proc. 11th WSEAS international conference on Electronics, Hardware, Wireless and Optical Communications*, and in *Proc. 11th WSEAS international conference on Signal Processing, Robotics and Automation, and proceedings of the 4th WSEAS international conference on Nanotechnology* (EHAC'12/ISPRA/NANOTECHNOLOGY'12), 2012, pp. 51-62.

[3] A. Enache, C.-A. Boiangiu, "Adaptive video streaming using residue hypercubes," in *Proc. 12th WSEAS International Conference on Circuits, Systems, Electronics, Control & Signal Processing* (CSECS '13), Budapest, Hungary, December 10-12, 2013, pp. 173-179.

[4] W.-S. Kim, H. M. Kim, "Residue sampling for image and video compression," *Visual Communications and Image Processing* 2005, *Proc. of the SPIE*, vol. 5960, pp. 12-18.

[5] L. Vandendorpe, B. Macq, "Optimum quality and progressive resolution of video signals," *Annales Des Télécommunications*, vol. 45, 1990, pp. 487-502.

[6] M. S. Kishore, K V. Rao, "A study of correlation technique on pyramid processed images," *Sadhana*, vol. 25, 2000, pp. 37-43.

[7] V. Swathi, M. Tech, K. A. Babu, "Low bit-rate image compression using adaptive down-sampling technique," *International Journal of Computer Technology and Applications*, vol. 15, 2002, pp. 1679-1689.

[8] M. Rabbani, R. Joshi, "An overview of the JPEG 2000 still image compression standard," *Signal Processing: Image Communication*, vol. 17, 2002, pp. 3-48.

[9] C.-A. Boiangiu, A. Olteanu, A. V. Stefanescu, D. Rosner, A. I. Egner. "Local thresholding image binarization using variable-window standard deviation response," in *Proc. 21st International DAAAM Symposium*, 20-23 October, 2010, Zadar, Croatia, pp. 133-134.

[10] X. Song, Y. Neuvo, "Image compression using nonlinear pyramid vector quantization," *Multidimensional Systems and Signal Processing*, vol. 5, 1994, pp. 133-149.

[11] C.-A. Boiangiu, B. Raducanu, "Effects of data filtering techniques in line detection," *Annals of DAAAM for 2008, Proceedings of the 19th International DAAAM Symposium*, Vienna, Austria, pp. 0125–0126.

[12] J. B. T. M. Roerdink, "Morphological pyramids in multiresolution MIP rendering of large volume data: survey and new results," *Journal of Mathematical Imaging and Vision*, vol. 22, 2005, pp. 143-157.

[13] R. Di Salvo, C. Pino, "Image and video processing on CUDA: state of the art and future directions," in *Proc. of the 13th WSEAS international conference on mathematical and computational methods in science and engineering*, 2011, pp. 60-66.