Efficient and Scalable On-Demand Data Streaming Using UEP Codes

Lihao Xu
Department of Computer Science
Washington University in St. Louis
lihao@cs.wustl.edu

ABSTRACT

In this paper, we propose and analyze a new multicast scheme for delivering on-demand streaming data using UEP (*Un-Equal Protection*) codes. The scheme allows an end user to join the multicast channel for a data stream at *any* time to play out the requested data stream from its beginning after a fixed amount of initial delay time. Resource usage of the scheme, including server computing bandwidth, network bandwidth and client's buffer space, is only determined by the original data stream length and the initial playout delay, but *independent* of either the number or the arrival pattern of individual end user requests. Thus the scheme is totally scalable with the number of end users, fully utilizing the data delivery efficiency of a multicast network. The scheme also uses resources efficiently, e.g., with an initial delay of 30 and 60 seconds respectively, multicasting a 2-hour video using this scheme only needs respectively about 5.5 and 4.8 times server computing bandwidth and network bandwidth of that for a single unicast delivery of the same original data stream. In addition, the scheme also tolerates packet loss during transmission, thus significantly reduces the cost of implementing a reliable multicast network layer to ensure delivery of all packets.

Keywords

on-demand streaming data, multicast, UEP code, efficiency, scalability

1. INTRODUCTION

Continuous streaming media is emerging as an increasingly popular data delivery form for many applications, such as entertainment (video, audio-on-demand), medical information service, distance learning, newscast, etc. However, streaming data is usually large and thus consumes a great deal of resources, including server computation and I/O bandwidth, network bandwidth, and end users' buffer space. A popular data, such as a blockbuster movie, or a lecture video during final exam preparation time, is usually requested by many users in a short time period. Thus if the data stream is delivered using unicast, i.e., via a dedicated channel for each request, the load on server and the load on network are both roughly proportional to the number of users during the playout time period. Due to its poor scalability, this huge consumption of resources makes the unicast type of delivery form implausible even with ever growing available network bandwidth and computing power. For example, if a usual full-size 2-hour video is requested by just 1000 users in a short time, one unicast needs a network bandwidth of 3 Mbps to 8 Mbps, then the total network bandwidth consumed ranges from 3 Gbps to 8 Gbps. Also the server's load is roughly 1000 times of one unicast load. So in this case, the resource consumption, both the network bandwidth and server computing bandwidth (i.e., the amount of data a server can process), is extremely expensive for delivering only one video. Thus it is economical, both for the server and the network, to adopt more scalable data distribution approaches rather than point-to-point unicast. One of such scalable approaches is to multicast data to a group of users requesting the same data. Using multicast, both the server load and the network bandwidth consumption are roughly the same as those for one unicast delivery. Because of its natural high scalability, multicast is particularly suitable for delivering data to a large group of users. While an internet wide multicast infrastructure is not fully supported yet, it is feasible and economical to set up a multicast communication environment within a relatively smaller organizational network, such as a campus-wide network, an institutional intranet, or even a residential cable network. Hence it is desirable to explore schemes of delivering streaming data that utilize multicast to improve both efficiency and scalability of data delivery.

However, multicast only works well for synchronous users, i.e., the users have to join the multicast group at the same time in order to share the same data stream. For most applications, users usually request the same streaming data at different times, i.e., user data requests are usually *asynchronous*. Extensive research has been done to resolve this issue [1, 2][6] [8]–[12][14, 16, 17, 21, 24]. One class of techniques is to *batch* users' requests issued at different times, and then use periodic multicast of the same data stream and exploit users' buffer spaces to synchronize users' data reception [2, 8, 12, 14]. The shortcoming of this kind of scheme is of course a large time delay of data delivery to the users. A trade-off between the data playout delay and consumption of resources (server, network and user buffer

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ACM Multimedia 2001 Sept. 30 - Oct. 5, 2001, Ottawa, Ontario, Canada
Copyright 2001 ACM 1-58113-433-9/01/09 ...$5.00.

space) has to be made. For example, if a 2-hour video is multicast every 10 minutes, then the resources consumption is about 12 times as that for one unicast and the worst initial delay for a particular user is close to 10 minutes. But if we want to reduce the initial delay to 1 minute, then the resources consumption has to increase another 10-fold.

Another class of techniques used to resolve the asynchronous issue is to use the so-called *pyramid* or *skyscraper* broadcasting method [1, 6, 10, 11, 16, 21, 24]. Using this method, the server multicasts different segments of the requested data to different channels or connections. These channels may have different bandwidths, or equivalently, different data rates are used for different channels on which different segments of the requested data are periodically multicast. A user may receive data from multiple channels simultaneously while playing out proper segments of data that have been accumulated in its local buffer. The key of this class of techniques is to decide how to partition the requested data into different segments and the rates by which they are multicast, depending on the arrival time pattern of users' requests. Of course, a trade-off has to be made between data playout delay and resource consumption. People also use hybrid of the above two classes of techniques. So far, assuming the users' requests arrive in a *Poisson* process, the lowest resource consumption with no initial data playout delay is proportional to logarithm of the *average* request arrival rate, the average number of requests arriving within some fixed time period [11]. This is much better than the delivery schemes based on point-to-point unicast, whose resource consumption is proportional to the number of requests.

However, the resource consumption using currently known schemes still depends on not only the number of user requests, but also the request arrival patterns. This leads to poor scalability in terms of number of users, which weakens the advantage of multicast. Also in designing and building a practical on-demand streaming data delivery system, it is usual not so easy to predict user request arrival patterns, thus resource consumptions in a real environment may further increase. Thus it is desirable to explore efficient on-demand data delivery schemes whose resource consumption only depends on the requested data length, but totally *independent* of the number of requests. Such schemes will completely scale with the number of requests (or users), and thus fully utilize the scalable nature of multicast.

All the currently known streaming-media-on-demand schemes have not addressed the reliability issue of delivery over multicast network, where it is assumed that the multicast network is a reliable network without packet loss. In practice, one disadvantage of multicast is the expensive cost of implementing a reliable multicast channel on top of an unreliable physical communication layer. If different receivers in a multicast group lose different data packets, the sender has to resend all of them either using unicasts or several multicasts. One common practice to reduce this cost is to use error control codes that enable forward error/loss correction/recovery (FEC) rather than the conventional retransmission-based ARQ (*automatic-repeat-request*) type of protocols [7, 13]. However, these schemes can only deal with delivery over multicast networks of *bulk* data rather than *streaming* data. One major difference between bulk data (such as a software file) and streaming data (such as a movie) is that a bulk data is consumed by users only after it has been delivered *in whole*, while a streaming data is consumed while it is *being* delivered. Thus all portions of a bulk data have *equal* importance in the sense that only after all the portions are delivered, is the combined whole bulk data consumed. On the other hand, different portions of a streaming data have *different* importance, since usually certain portions of data are more needed than the others at a specific time. For example, a software is executed only after the whole software file has been downloaded, while ideally a movie is played out while it is being delivered. Thus from timely data delivery point of view, all portions of a software are equally important, while the beginning portions of a movie are more important than the later ones (assuming most people want to watch a movie from its beginning). In fact we can regard streaming data as a more general form of bulk data, since streaming data can certainly be consumed after it has been downloaded to a local buffer in whole. As will be seen in later sections, error control codes used in bulk data multicasting are just a special instance of the one proposed in this paper with equal protection degrees of all symbols.

In this paper, we propose a new multicast scheme for delivering streaming data efficiently, using UEP (*UnEqual Protection*) codes, a class of error control codes. Distinguished from all other existing schemes, our scheme multicasts a stream of UEP-encoded form of the original data, rather than the original data stream itself. The encoded data stream is multicast cyclically through only *one* logical channel. An end user can join the multicast group at *any* time by listening to the multicast channel during the data transmission period, and after a small fixed time (which could be as low as zero) delay, the user can play out the data from its beginning. This solves the very issue of the asynchronous nature of user requests. More appealingly, the resource consumption of our scheme is a *constant* times of that for a single unicast of the original data. This constant only depends on the data stream length and the initial playout delay designed, but totally *independent* of the number of end users. Thus the scheme is *fully scalable* to any number of end users. Again, as already pointed out above, though the so-called digital fountain type schemes [7, 13] also use error control codes to improve multicast efficiency by reducing or eliminating retransmission of lost packets, they are only applicable to bulk data. The scheme proposed in this paper is applicable to *both* streaming data *and* bulk data.

Before discussing the scheme in detail, we show its high efficiency using streaming a 2-hour video as an example. For a 2-hour video with 30 frames per second, if no additional playout time delay is allowed (excluding the connection setup time and the delay for the first frame which cannot be avoided by any scheme, thus hereafter we will omit the word *additional* when we refer to initial delay, as is clear in the context), using our scheme, a user can play out the video at any time from the beginning, while the whole data delivery system consumes only less than 12.9 times of resources (including server computation loads and network bandwidth) for a single unicast of the original video and user's buffer space of about 37% of the original video size. If a user allows an initial playout delay of just 1 second, the resources needed dramatically drops to 8.87 times of that for a unicast. If a user allows 30 seconds initial playout delay, then multicast resources needed reduce to about 5.5 times of that for a unicast, with roughly the same buffer space at the user's side. If the initial playout delay is allowed to increase to 1 minute, then the multicast resources needed further reduce to about 4.8 times of that for a single unicast and the buffer space needed at the user's side remains almost the same. Again, all the data shown above are independent of the number of requests for the original data stream, and they are listed in Table 1.

d	0	1	30	60
R	12.859	8.87	5.484	4.796
C	36.788%	36.790%	36.941%	37.095%

Table 1: Resource Consumption vs. Initial Playout Delay for Multicasting a 2-hour Video, where d is the initial playout delay in *seconds*, R is the *normalized* backbone network bandwidth/server computing bandwidth, and C is the *normalized* client buffer space needed.

The key of this scheme is to use a desirable property of UEP codes: different original data symbols can be retrieved from different numbers of symbols from a continuous UEP encoded codeword stream. As a class of error control codes, UEP codes also have erasure/error recovery capability. Thus by employing UEP codes, our scheme can tolerate packet loss during transmission. This greatly reduces the cost of implementing a reliable multicast network. The so-called *Harmonic Broadcasting* scheme and its variations [16, 17, 21] are just a special implementation of our scheme, without the packet loss recovery capability.

In Section 2, we will briefly introduce the UEP codes. Section 3 describes the new streaming data multicast scheme; analyzes its efficiency in resource usages, including server computing bandwidth, network bandwidth and client's buffer space. Section 3 also suggests balances between initial data stream playout delay and resource consumption. Section 4 concludes the paper with a few open problems.

2. UEP (UNEQUAL ERROR PROTECTION) CODES

Unequal Error Protection Codes, or *UEP* codes, are a class of error control codes. Error control codes are a mathematical means of adding proper redundant information to original data to provide some degree of protection against data loss and/or corruption during data transmission and/or storage [18, 25]. A usual (N, K) block code *encodes* an original message of K data symbols into a *codeword* of N data symbols of the same size. A data symbol here is a general data unit of certain size: it could be a bit, a byte, a packet or a frame. Proper error control property of such a code ensures that the original K data symbols can be recovered from *any* M data symbols of its codeword. The recovery process is called the *erasure* decoding process. (An error symbol is a data symbol that is corrupted during delivery, and an erasure symbol is a data symbol that is lost while being delivered.) Obviously it is necessary that $M \geq K$, which is one form of the well-known Singleton Bound in coding theory [18]. The code is called an *MDS* code if $M = K$ [18]. Thus for a usual (N, K) code, all the K original data symbols are of the same importance, and they, *as a whole message*, can be recovered from *any* M symbols of the corresponding codeword. This property is the foundation of all the schemes efficiently delivering bulk data over an unreliable multicast network, such as the schemes proposed in [7, 13].

Unlike conventional error control codes, for a UEP code, certain data symbols of its codeword are protected against a greater number of *errors* (or *erasures*) than others [19, 22, 23, 4]. Thus different data symbols of a UEP codeword have different error protection degrees. The error protection degree of a data symbol in a codeword can be measured by the *maximum* number of the errors and/or erasures the codeword can tolerate to retrieve this symbol. Equivalently, we can also use the *minimum* number of data symbols needed to retrieve a data symbol in a codeword to measure the error protection degree of this data symbol. For a message m with n data symbols, if the error protection degree of its ith symbol is L_i $(1 \leq i \leq n)$, and it is encoded with a UEP code C of N symbols, then *any* L_i symbols of its codeword are sufficient to retrieve the ith symbol in the original message m. If $L_i = M$ holds for $1 \leq i \leq n$ where M is a certain constant, then this UEP code reduces to a conventional (N, n) error control code described above. On the other hand, a UEP code of length N with protection degrees $L_i's$ can be constructed from (N, L_i) MDS codes [23, 4], especially from MDS codes that can be encoded and decoded with efficient computations, such as MDS array codes [26, 27].

Since different portions of most multimedia streaming data have different importance, such as I-frames and P-frames of MPEG data, different layers of multi-resolution description of media data, UEP codes have naturally been proposed to protect data loss and/or corruption during transmission to improve the quality of data delivery in many applications, such as [15, 20]. The main goal of the UEP codes in these schemes is to deal with the data loss/error during transmission. In contrast, the scheme proposed in this paper uses the UEP codes mainly to address the asynchronous issue of user requests over multicast networks with efficient use of system resources. Data loss and error during transmission are naturally overcome by the inherent error protection capability of the UEP codes.

Before describing our scheme, we discuss one more property of UEP code that directly decides the designing parameters of the UEP codes used in our scheme. The following theorem states the relation among the error protection degrees (L_i's) of different symbols in a message, and it can be derived from the results developed in [4]:

THEOREM 1. *For a message m with n symbols, if there exists a UEP code such that the error protection degree of the ith symbol in the original message m is L_i $(1 \leq i \leq n)$, then $\sum_{i=1}^{n} 1/L_i \leq 1$.*

Notice that the above inequality is *independent* of the UEP codeword length N. Also when $L_i = M$ for $1 \leq i \leq n$, the above inequality is just another form of the well-known Singleton Bound in error control coding theory [18]. Thus we call the bound in the above theorem the *Generalized*

Singleton Bound.

Finally we give a detailed example of a UEP code that meets the Generalized Singleton Bound:

EXAMPLE 1. *Suppose the original streaming data message m has 3 symbols of equal size: a, b and c, i.e., $m = abc$. Now partition symbol a into 6 sub-symbols of equal size: $a = a_1 \cdots a_6$, b into 9 sub-symbols $b = b_1 \cdots b_9$, and c into 6 sub-symbols: $c = c_1 \cdots c_6$.*

Now apply the (6,2) B-Code [27] on a to get a codeword of a: $A = A_1 \cdots A_6$, where
$A_1 = a_1, a_2 + a_3, a_4 + a_6,$
$A_2 = a_2, a_3 + a_4, a_5 + a_1,$
$A_3 = a_3, a_4 + a_5, a_6 + a_2,$
$A_4 = a_4, a_5 + a_6, a_1 + a_3,$
$A_5 = a_5, a_6 + a_1, a_2 + a_4,$
$A_6 = a_6, a_1 + a_2, a_3 + a_5,$
and + is the simple bit-wise binary exclusive or (XOR) operation. Notice here each A_i is only 1/2 size of a.

The (6,2) B-Code is an MDS code, i.e., all the original data symbols a_1 through a_6 can be recovered from any 2 of the 6 symbols A_1 through A_6 of the codeword A. For example, from A_1 and A_2, a_1 and a_2 are immediately recovered. Then a_3 is recovered from a_2 (from A_2) and $a_2 + a_3$ (from A_1): $a_3 = a_2 + (a_2 + a_3)$. Similarly, $a_5 = a_1 + (a_5 + a_1)$. Subsequently, a_4 and a_6 can be recovered: $a_4 = a_3 + (a_3 + a_4)$ and $a_6 = a_4 + (a_4 + a_6)$. The above decoding process only uses the bit-wise XOR +, thus it is very efficient to compute. It is easy to verify that all the other pairs of A_i's can recover the original a_i's. (Interested readers are encouraged to do so!)

Then apply a modified (6,3) RS-Code [28] on b to get a codeword of b: $B = B_1 \cdots B_6$, where
$B_1 = b_1, b_2, b_3,$
$B_2 = b_1 + b_3 + b_4, b_2 + b_4 + b_5, b_2 + b_3 + b_6,$
$B_3 = b_6 + b_7, b_4 + b_6 + b_8, b_5 + b_9,$
$B_4 = b_3 + b_6 + b_7, b_1 + b_4 + b_7 + b_8, b_2 + b_5 + b_9,$
$B_5 = b_4 + b_6 + b_9, b_4 + b_7,$
$B_6 = b_7, b_8, b_9,$
again + is the XOR operation. It is easy to see that each B_i is 1/3 size of b. Again, the modified (6,3) RS-Code is an MDS code, i.e., b_1 through b_9 can be recovered from any 3 symbols of the codeword B.

Next, just let $C_i = c_i$ for $i = 1, \cdots, 6$, thus each C_i is 1/6 size of c.

Finally, construct a UEP codeword of the original message m: $U = U_1 \cdots U_6$, where $U_i = A_i B_i C_i$ for $i = 1, \cdots, 6$. It is not hard to verify that 1) each U_i is of the same size as a (or b or c); 2) a, b and c can be recovered from any 2, 3 and 6 different U_i's respectively, i.e., the protection degrees of the original data symbols a, b and c are $L_a = 2$, $L_b = 3$ and $L_c = 6$ respectively. Thus this UEP code meets the Generalized Singleton Bound: $1/L_a + 1/L_b + 1/L_c = 1$. This is not surprising, since all the component codes used A, B and C are MDS codes which meets the Singleton Bound.

It is worth pointing out the only operation needed for encoding (obtain the codeword from the original message) and decoding (recover the original message from certain symbols of the codeword) is simple + (XOR). Thus both encoding and decoding are very efficient to compute. This comes from the computation efficiency of the component codes used, namely the B-Code and the modified RS-Code [27, 28].

□

3. MULTICASTING A DATA STREAM USING UEP CODES

The UEP codes provide an efficient means of delivering streaming data. For a streaming data with n symbols, its ith symbol should be retrieved and then played out before the $(i+1)$th symbol is delivered. This can be realized by using a proper UEP code: let the ith symbol of the original data stream have an error protection degree of L_i $(1 \leq i \leq n)$, then we just need $L_i \leq L_{i+1}$ for $1 \leq i \leq n-1$. The UEP code guarantees the retrieval of the ith symbol of the original data as long as any L_i symbols of its codeword are received. Thus when the UEP codeword (with N symbols) of a data stream is being *multicast* to a group of end users, a new user can join the multicast group at any time during this multicast period. Then the new user can retrieve the ith symbol of the original streaming data once L_i symbols from the UEP codeword are received. Thus a *single* UEP encoded data stream enables multiple users to play out the original data from its beginning at *any* specific moment.

Now we describe our new multicast scheme.

3.1 The Scheme

First the streaming data server encodes an original data of n symbols into a UEP codeword of N symbols such that the error protection degree of the ith symbol of the original data is L_i for $1 \leq i \leq n$. Upon the first request for the original data, the server continuously sends the stream of its UEP codeword to a multicast channel in a cyclic fashion. This multicast process ends only after all the users in the group finish receiving the data stream.

A new user requesting the original streaming data first asks the server for the multicast group identifier, e.g., the multicast IP address, of the corresponding encoded UEP stream. It then joins the multicast group, receives the UEP data stream symbols and stores them at its local buffer space. Once the number of data symbols in its buffer space reaches L_i, the user retrieves (decodes) the ith symbol of the original data stream and plays it out. The user leaves the multicast group when its data request is fulfilled.

It can be readily seen that the above scheme greatly reduces server's load, since the server only needs to multicast one encoded UEP data stream all the time, regardless the number of the end users requesting the original data. Also the network bandwidth consumption is reduced dramatically too. If a normal playout of the original data needs a unicast network bandwidth B (which we refer to as the *base bandwidth* hereafter) and the initial playout is d original data symbols, then the peak network bandwidth needed to play out the original streaming data at its original (normal) playout rate is

$r_{peak}B$, where $r_{peak} = max(\frac{L_i}{i+d})$ for $1 \leq i \leq n$, and the average network bandwidth is $r_{ave}B$, where $r_{ave} = L_n/(n+d)$. (Note here for simplicity of mathematical representations, we measure the initial playout delay by the number of original symbols, which can be easily converted to the corresponding playout time.) Again both the peak and the average network bandwidths are independent of the number of the end users. Thus we see this scheme has the natural scalability in terms of the number of users.

In addition, the UEP codes inherently tolerate some degree of data symbol loss during the transmission. Thus the multicast channel does *not* need to be a reliable one, as long as a delivered data symbol carries a symbol index, which can be easily implemented using some common techniques, such as sequence numbers. This greatly improves communication efficiency, since the cost of implementing a reliable multicast channel without data loss is extremely expensive. Thus our goal of efficiently and economically implementing a reliable multicast over unreliable communication medium is achieved.

Next we briefly analyze resources consumption of our scheme, and show the efficiency of the scheme.

3.2 Resources Consumption

Let's start with the network bandwidth consumption. The similar analysis can be readily applied to the computational load at the server's side.

3.2.1 Network Bandwidth and Server Computation Load

As shown in the previous section, the peak network bandwidth needed to play out the original data stream at its normal rate is $r_{peak}B$, where $r_{peak} = max(\frac{L_i}{i+d})$, and d is the initial playout delay measured by the number of original data stream symbols. It is easy to derive that in order to *minimize* r_{peak}, we need to set $\frac{L_i}{i+d} = R$ for all i's, where R is a constant. Thus we need to have $L_i = R(i+d)$. Together with Theorem 1 (the Generalized Singleton Bound), it is easy to obtain that $R \geq H_{n+d} - H_d$, where H_m is the *harmonic function* $H_m = \sum_{i=1}^{m} 1/i$. Thus the *minimal normalized* network bandwidth needed is $R = H_{n+d} - H_d$. In this case, the UEP code employed meets the Generalized Singleton Bound, and the network bandwidth needed to play out each original data symbol is a *constant*, since $L_{i+1} - L_i = R$ for all $i's$. This is a very desirable property for delivery of streaming data from the network QoS point of view.

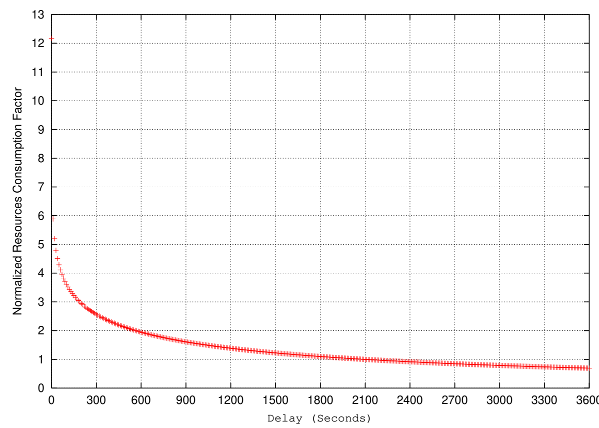
It is easy to prove that

THEOREM 2. $1 + \frac{1}{2}log_2 l \leq H_l \leq 1 + log_2 l$, *thus* $H_l \approx 1 + \alpha log_2 l$, *where* $\frac{1}{2} \leq \alpha \leq 1$.

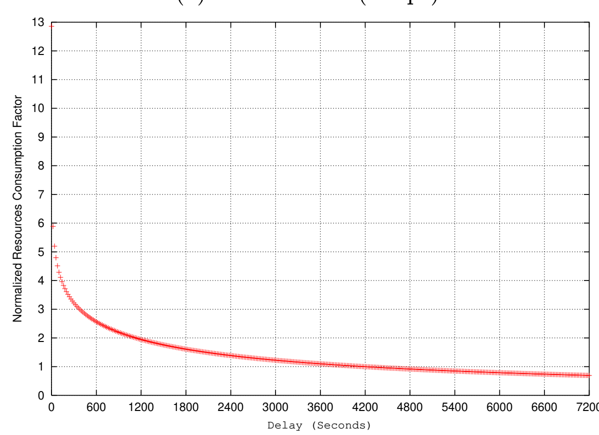
(It can be readily shown that $\lim_{l \to \infty} H_l = lnl$.)

Thus we see that for our proposed scheme, its normalized network bandwidth (or server computation load) is approximately $1 + \alpha log_2(1 + \frac{n}{d})$, where $\frac{1}{2} \leq \alpha \leq 1$; i.e., it is only *logarithm* of the ratio of the original data stream length and the playout delay measured by the number of original data symbols.

Figures 1(a)(b) show the normalized network bandwidth (or server computation load) vs. the initial playout delay allowed for delivering a 1-hour video and a 2-hour video respectively, using our scheme. The original playout rate for the both videos is 30 fps (frames per second), and it is natural to use one frame as a data symbol. Thus $n = 30T$ and $d = 30t$, where T and t are the video length and the designed initial playout delay *time* in seconds.



(a) 1-hour video (30 fps)



(b) 2-hour video (30 fps)

Figure 1: Normalized Server Load and Network Bandwidth vs. Playout Delay

Careful readers must have observed: when the initial delay increases to certain threshold (2100 and 4200 seconds in Figures 1(a) and (b) respectively), the normalized network bandwidth needed drops to less than 1. The reason is simple: the initial delay reduces network bandwidth usage, since a 1-hour or 2-hour video is now delivered using much more than 1 hour (2100+3600 seconds) or 2 hours (4200+7200 seconds).

If the original data is transmitted at a variable bit rate (VBR), we can simply adjust the error protection degrees L_i's in the UEP codeword to maintain a constant transmission rate for the UEP codeword stream. Thus a constant network bandwidth consumption can still be maintained. Let the playout rate of the ith symbol of the original data stream be R_i, in order to maintain a constant rate of the UEP codeword stream, we only need

$$\frac{L_{i+1} - L_i}{R_{i+1}} = \beta, \quad 1 \le i \le n$$

where the constant β is the network bandwidth needed for the UEP codeword stream. For a given series of R_i's, an optimal (minimal) β can be obtained which satisfies the equality in Theorem 1. Thus our scheme enables constant network bandwidth consumption for a data stream that is originally CBR or VBR. This of course is a preferable property for QoS.

3.2.2 *Client's Buffer Space*

Now we briefly discuss the buffer space that a client (user) needs locally for playing out a data stream at its original playout rate.

Certain data symbols of the UEP codeword stream have to be buffered at the user's storage space before the original stream can be played out. A naive buffering scheme is simply to buffer all the L_n symbols of the UEP codeword. However, in this case, for an original CBR stream, $L_n = R(n+d) = (H_{n+d} - H_d)(n+d)$. Thus if the original data stream length is M, then buffer space needed is $(1+\frac{d}{n})(H_{n+d} - H_d)M$, or roughly $(H_{n+d} - H_d)M$. By Theorem 2, the client's buffer space normalized over the original data size is approximately $1 + \alpha log_2(1+\frac{n}{d})$, where $\frac{1}{2} \le \alpha \le 1$, which is too expensive. Thus we seek a more efficient buffering scheme that needs much less buffer space.

We observe that in order for a UEP code to work, a fraction of each symbol of the original data stream is carried in each of the corresponding UEP codeword symbols. Once the ith symbol of the original data stream is retrieved, the information related to this symbol carried in all the subsequent UEP codeword symbols no longer needs to be buffered. Thus buffer space can be reduced significantly. Using this observation, it is easy to calculate the buffer space needed at a specific time after a client starts receiving the multicast UEP encoded data stream.

To minimize the network bandwidth usage, we need to set normalized network bandwidth $R = H_{n+d} - H_d$. In this case, the UEP code meets the Generalized Singleton Bound and $1/l_i$ of each UEP codeword symbol contains the information of the ith original data symbol. Thus the buffer space needed for retrieving the $(i+1)$th original symbol after the retrieval of ith original symbol is $(1 - \sum_{j=1}^{i} \frac{1}{l_j})l_{i+1}$ symbols. So for an original data stream with n symbols, let S_i be the buffer space needed between the retrieval of the ith and the $(i+1)$th original data symbols, normalized over the original data stream length, then

$$S_i = \frac{1}{n}(1 - \sum_{j=1}^{i} \frac{1}{l_j})l_{i+1}, \quad 1 \le i \le n-1$$

Using the substitution $l_i = R(i+d) = (H_{n+d} - H_d)(i+d)$, we get

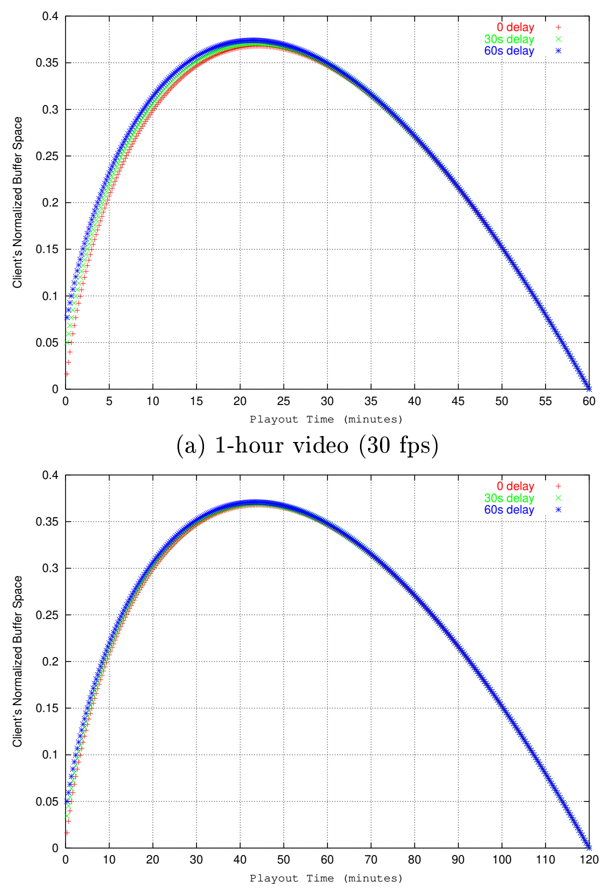
$$S_i = \frac{(H_{n+d} - H_{i+d})(i+d+1)}{n}, \quad 1 \le i \le n-1$$

On the other hand, the normalized buffer space needed before playing out the 1st original data symbol is simply $S_0 = (H_{n+d} - H_d)(d+1)/n = l_1/n$, thus the maximal normalized buffer space needed S is

$$S = \max_{0 \le i \le n-1} \frac{(H_{n+d} - H_{i+d})(i+d+1)}{n}$$

So for an original data stream with n symbols, once the initial playout delay d is determined, it is easy to calculate the buffer space needed to play out each original data symbol and the buffer space needed to play out the whole original data stream.

Figure2(a) and Figure 2(b) respectively show the normalized buffer space needed at a client's side for playing out a 1-hour video and a 2-hour video vs. the playout time. Again, each video plays at the rate of 30 frames per second, and additional initial playout delays are 0, 30 and 60 seconds respectively.



(a) 1-hour video (30 fps)

(b) 2-hour video (30 fps)

Figure 2: Client's Normalized Buffer Space vs. Playout Time

It is easy to see that at the beginning, the client needs to buffer data for future playout, thus the buffer space needed increases with the time. Then at certain point, the client's local buffer has accumulated enough data for current playout together with incoming data, i.e., data stream is mainly consumed from the accumulated ones from the local buffer, hence the local buffer space needed decreases with the time, and eventually drops to zero after the whole original data stream has been played out.

Buffer space for original VBR data stream can be calculated in a similar way once L_i's are decided by the original data rates R_i's.

3.2.3 Client's Network Bandwidth

Though the UEP encoded data stream needs to be sent to one logical multicast channel with a network bandwidth depending on the initial playout delay as discussed previously, in practice, several parallel multicast channels can certainly be used to form the single logical channel with the required bandwidth. Different portions of the UEP encoded data can be sent over different channels, and clients (users) can tap to single or multiple channels at the same time as needed. Similar observation on a client's buffer space needed can be made on the actual network bandwidth a specific client consumes as original data is being retrieved and played out: once a client plays out the ith original data symbol, it does *not* need to listen the channel(s) where those encoded data symbols containing the ith original data symbol are being multicast. So again it is easy to calculate the actual network bandwidth a client needs at a specific playout time. The normalized network bandwidth a client needs to consume between the retrieval of the ith and $(i+1)$th original data symbols is $C_i = R(1 - \sum_{j=1}^{i} \frac{1}{l_j})$, using the substitutions $l_i = R(i+d)$ and $R = H_{n+d} - H_d$, we get

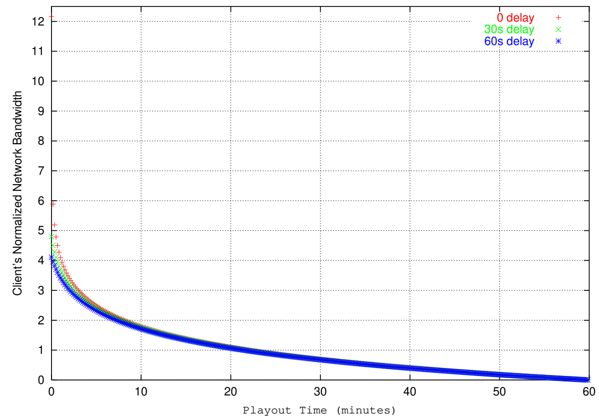
$$C_i = H_{n+d} - H_{i+d}, \quad 0 \le i \le n-1$$

In Figures 3(a)(b), we show a client's network bandwidth normalized over the base bandwidth for 1-hour and 2-hour videos respectively. Each figure, again, shows three different initial playout delays: 0, 30 and 60 seconds, and each video has the playout rate of 30 frames per second.

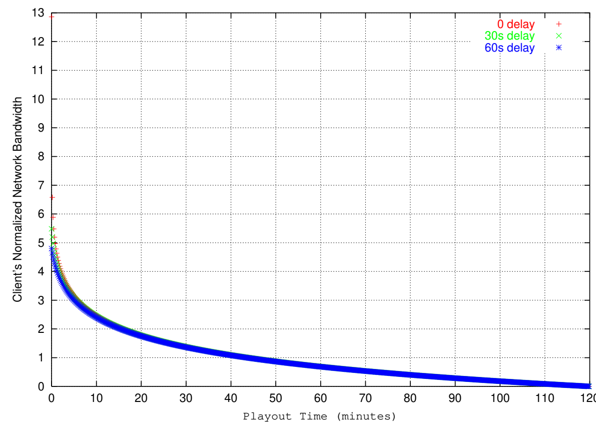
Here we see the network bandwidth needed by a client decreases as the playout time goes by, since less and less data needs to be delivered to the client. It is interesting to point out that in this scheme, after some playout time, the incoming network bandwidth an individual client needs drops to less than the base bandwidth because of the use of local buffer. Further analysis can easily show that the average network bandwidth a client needs in the scheme is exactly the same as the base bandwidth, if the employed UEP code meets the Generalized Singleton Bound. This suggests that it is feasible to use this scheme even for a client with lower incoming bandwidth. The trade-off is just to increase the initial playout delay at the client's side. Thus clients with different incoming network bandwidths can be served by a single UEP encoded data stream, as long as the backbone network bandwidth from the server is high enough.

3.3 Additional Initial Playout Delay

As already shown, the normalized backbone network bandwidth is determined by the additional initial playout delay d: $R(d) = H_{n+d} - H_d$, and $R(d)$ is a *strictly decreasing* function of d. Thus to save the backbone network bandwidth and server computing bandwidth, it is preferable to set d as big as possible, as long as the initial delay is tolerable by the end users. Once the initial delay d is decided, the UEP



(a) 1-hour video (30 fps)



(b) 2-hour video (30 fps)

Figure 3: Client's Normalized Network Bandwidth vs. Playout Time

code can be decided accordingly and thus the encoded data stream is also fixed.

On the other hand, in many applications, individual end users have different (last mile) incoming networks connected to them, thus they have different incoming network bandwidths. For an end user with its normalized incoming network bandwidth W, if $W \ge R(d)$, then its initial playout delay is d original data symbols. However, if $W < R(d)$, then in order to play out the original data stream smoothly and continuously, it needs to buffer more than l_1 encoded data symbols before it can start playing out the original data stream, i.e., it needs more additional initial delay. Let $D(W,d)$ be the additional initial delay (measured by the number of original data symbols) for an end user with its incoming network bandwidth W to play out a data stream with the designed delay d, then for $W < R(d)$, we have

$$D(W,d) = d + (\frac{1}{W} - \frac{1}{R(d)})l_n$$

Again, using the substitutions $l_n = R(d)(n+d)$ and $R(d) = H_{n+d} - H_d$, $D(W,d)$ can simplified to

$$D(W,d) = \frac{f(d)}{W} - n$$

where $f(d) = (n+d)(H_{n+d} - H_d)$. But $f(d-1) - f(d) =$

$\sum_{i=d}^{n+d-1}(\frac{1}{d}-\frac{1}{i})>0$, i.e., $f(d)$ is a strictly decreasing function of d, thus $D(W,d)$ too is a strictly decreasing function of d when $W<R(d)$. Combining the case of $W\geq R(d)$, the initial playout delay of a client with its normalized incoming network bandwidth W is

$$D(W,d)=\begin{cases}(n+d)R(d)/W-n & W<R(d)\\ d & W\geq R(d)\end{cases}$$

where $R(d)=H_{n+d}-H_d$ and the original data stream has n symbols and d is the designed initial playout delay. Again all delays are in terms of the number of original data symbols.

This indicates that the larger the designed initial playout delay d is, the less resources (server computing bandwidth and backbone network bandwidth) need to be consumed, and the less initial playout delay a client with a low incoming network bandwidth has. The only disadvantage a large designed initial playout delay brings is that clients with high incoming bandwidth also have to suffer the large delay even if they can afford high network bandwidth. Thus, in practice, the designed initial playout delay d should be decided by the clients' needs. Usually, for a video of 1 to 2 hours, 30 to 120 seconds would be reasonable for the designed initial playout delay.

4. CONCLUSIONS

As continuous streaming media is emerging as an increasingly popular data delivery form for many applications, it is important to deliver massive amount of streaming data to large groups of users in an efficient and scalable way by effectively utilizing system resources: computing bandwidth, network bandwidth and storage space. In this paper, we propose an efficient and fully scalable scheme of delivering streaming data over multicast networks. Our scheme utilizes nice properties of error control codes, particularly UEP codes with MDS array codes as their building blocks. The scheme also tolerates packet loss during transmission, thus further reduces multicast cost.

In this scheme, users have to play out the original data stream from its beginning. In some applications, it is desirable to have the *fast seek* feature, i.e., users are able to access a random point of data stream instead of the beginning with a fixed playout delay. One open problem is: what is the minimal resource (mainly server computing bandwidth and backbone network bandwidth) requirement to allow *any* user to do fast seek at any time, with a fixed seek delay D? And how to encode the data stream to achieve the fast seek feature? Further more, how to multicast *interactive* streaming data such as multimedia games? Another research problem is how to apply to UEP coding idea to efficiently deliver streaming data over a content distribution network (*CDN*) that is not multicast supported.

5. REFERENCES

[1] C. C. Aggarwal, J. L. Wolf and P. S. Yu, "A Permutation-Based Pyramid Broadcasting Scheme for Video-on-Demand Systems", *Proc. of the IEEE International Conference on Multimedia Computing and Systems*, 118-126, 1996.

[2] C. C. Aggarwal, J. L. Wolf and P. S. Yu, "On Optimal Batching Policies for Video-on-Demand Storage Servers", *Proc. of the IEEE International Conference on Multimedia Computing and Systems*, 1996.

[3] D. Aksoy and M. Franklin, "Scheduling for Large-Scale On-Demand Data Broadcasting", *IEEE INFOCOM*, Mar. 1998.

[4] A. Albanese, J. Blömer, J. Edmonds, M. Luby and M. Sudan, "Priority Encoding Transmission", *IEEE Trans. on Information Theory*, 42(6), 1737-1744, Nov. 1996.

[5] A. Bar-Noy, R. Bhatia, J. Naor and B. Schieber, "Minimizing Service and Operations Cost of Periodic Scheduling", *Proc. of the 9th ACM-SIAM SODA*, 1998.

[6] A. Bar-Noy and R. Ladner, "Efficient Algorithms for Optimal Stream Merging for Media-on-Demand", Computer Science Technical Report, University of Washington, *www.cs.washington.edu/homes/ladner/papers.html*

[7] J. Byers, M. Luby, M. Mitzenmacher and A. Rege. "A Digital Fountain Approach to Reliable Distribution of Bulk Data", *Proc. of the ACM SIGCOMM'98*, 56-67, Sep. 1998.

[8] Y. Cai, K. Hua and K. Vu, "Optimizing Patching Performance", *Proc. of the 7th ACM International Multimedia Conference*, 211-214, 1999.

[9] S. Carter, D. Long and J. Pâris, "Video-on-Demand Broadcasting Protocols", *Multimedia Communications* (J. Gibson Eds), Ch. 11, 179-189, Academic Press, 2001.

[10] D. Eager and M. Vernon, "Dynamic Skyscraper Broadcasts for Video-on-Demand", *Proc. of 4th Workshop on Multimedia Information Systems*, Sep. 1998.

[11] D. Eager, M. Vernoni and J. Zahorjan, "Minimizing Bandwidth Requirements for On-Demand-Data Delivery", *Proc. of the 5th International Workshop on Advances in Multimedia Information Systems*, 80-87, 1999.

[12] L. Gao, D. Towsley and J. Kurose, "Efficient Schemes for Broadcasting Popular Videos", *Proc. of International Workshop on Network and Operating System Support for Digital Audio and Video*, Jul. 1998.

[13] J. Gemmell, J. Gray and E. Schooler, "Fcast multicast file distribution", *IEEE Network*, 14(1), 58-68, Jan. 2000.

[14] K. Hua and S. Sheu, “Skyscraper Broadcasting: A New Broadcasting Scheme for Metropolitan Video-On-Demand Systems”, *Proc. of ACM SIGCOMM'97*, Sep. 1997.

[15] U. Horn, K. Stuhluller, M. Link, and B. Girod, “Robust Internet video transmission based on scalable coding and unequal error protection”, *Image Com.*, 15(1-2), 77-94, Sept. 1999.

[16] L. Juhn and L. Tseng, “Harmonic Broadcasting for Video-on-Demand Service”, *IEEE Trans. on Broadcasting*, 43(3), 268-271, 1997.

[17] L. Juhn and L. Tseng, “Enhanced Harmonic Data Broadcasting abd Receiving Scheme for Popular Video Service”, *IEEE Trans. on Consumer Electronics*, 44(2), 343-346, May 1998.

[18] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error Correcting Codes*, Amsterdam: North-Holland, 1977.

[19] B. Masnick and J. Wolf, “On Linear Unequal Error Protection Codes”, *IEEE Trans. on Information Theory*, 3(4), 600-607, Oct. 1967.

[20] A.E. Mohr, E.A. Riskin and R.E. Ladner, “Unequal Loss Protection: Graceful Degradation of Image Quality Over Packet Erasure Channels Through Forward Error Correction”, *IEEE Journal of Selected Areas in Communications Special Issue on Error-Resilient Image and Video Transmission*, 18(6), 819-828, 2000.

[21] J. Pâris, S. W. Carter and D. D. E. Long, “A Low Bandwidth Broadcasting Protocol for Video on Demand”, *Proc. of the 7th International Conference on Computer Communication and Networks*, 609-617, Oct. 1998.

[22] W. J. Van Gils, “Two Topics On Linear Unequal Error Protection Codes: Bounds On Their Length And Cyclic Code Classes”, *IEEE Trans. on Information Theory*, 29(6), 866–876, Nov. 1984.

[23] W. J. Van Gils, “Linear Unequal Error Protection Codes from Shorter Codes”, *IEEE Trans. on Information Theory*, 30(3), 544– 546, May. 1984.

[24] S. Viswanathan and T. Imielinski, “Metropolitan Area Video-on-Demand Service Using Pyramid Broadcasting”, *ACM Multimedia Systems Journal*, 4(3), 197-208, 1996.

[25] Stephen B. Wicker, *Error Control Systems for Digital Communication and Storage*, Prentice-Hall Inc., 1995.

[26] L. Xu and J. Bruck, “X-Code: MDS Array Codes with Optimal Encoding”, *IEEE Trans. on Information Theory*, 45(1), 272-276, Jan., 1999.

[27] L. Xu, V. Bohossian, J. Bruck and D. Wagner, “Low Density MDS Codes and Factors of Complete Graphs”, *IEEE Trans. on Information Theory*, 45(6), 1817-1826, Sep. 1999.

[28] L. Xu, “Maximizing Burst Detection and Correction Capability of MDS Codes”, submitted to *IEEE Trans. on Information Theory*, May 2001.
Online at: *www.cs.wustl.edu/˜ lihao/tech.html*