A Transpositional Redundant Data Update Algorithm for Growing Server-less Video Streaming Systems

T. K. Ho and Jack Y. B. Lee
Department of Information Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong
Email: {tkho2@ie.cuhk.edu.hk, jacklee@computer.org}

Abstract

Recently, a new server-less architecture is proposed for building low-cost yet scalable video streaming systems. In this architecture, video blocks are distributed among user hosts and these hosts cooperate to stream video blocks to one another. To improve reliability, data and capacity redundancy are introduced to sustain node failures. However, the data placement as well as the redundant data in the system will need to be updated whenever new nodes join the system. Results show that the redundancy update overhead is very significant and even exceeds that in data reorganization. In this study, we present a new Transpositional Redundant Data Update algorithm that takes advantage of the structure of Reed-Solomon Erasure Correction codes and employs a special encoding scheme to significantly reduce the redundancy update overhead, especially when updates are performed in batch.

1. Introduction

Peer-to-peer and distributed computing has shown great potentials in high-performance computing applications. Apart from computational problems, data and I/O-intensive applications can also benefit from the inherent scalability offered by distributed architectures. One such architecture, called server-less video streaming architecture, recently proposed by Lee and Leung [1] adopted this completely decentralized approach to eliminate the need for costly high-capacity video servers.

Unlike conventional video streaming systems built around the well-understood client-server model, a server-less video streaming system is built entirely from user hosts. Video blocks are distributed among these user hosts which then cooperate to stream video blocks to one another for playback. Lee and Leung [1] showed that this server-less architecture is easily scalable to hundreds of user hosts using off-the-shelf computers and network switches. Moreover, by incorporating data and capacity redundancy into the system, one can even achieve system-level reliability comparable to or even exceeding those of dedicated video servers [2].

The study by Lee and Leung [1] is focused on the scalability and feasibility of the server-less architecture. They did not, however, address the practical problem of system growth when new user hosts join the system. Specifically, as video blocks are distributed among user hosts, these data will need to be redistributed to newly joined hosts to utilize their storage and streaming capacity. This data reorganization problem has been investigated by Ghandeharizadeh and Kim [5], Goel et al. [6], and Ho and Lee [3] respectively. On the other hand, the redundant data that are themselves computed from the video blocks will also need to be updated according to the change in data placement. This redundant data update problem has recently been studied by Ho and Lee [4], who proposed a Sequential Redundant Data Update (SRDU) algorithm.

In this study, we extend the work of Ho and Lee [4] in two ways. First, we propose a new Transpositional Redundant Data Update (TRDU) algorithm to further exploit the special structure in computing the updated redundant data in batched update. Results show that TRDU can significantly reduces overhead compared to SRDU for large batch sizes. Second, we optimize the algorithms for use in systems with multiple redundant nodes.

In the next section, we first briefly review the server-less architecture and previous works on data reorganization and redundant data update.

2. Background

We present an overview of the server-less architecture

This work was supported in part by the Hong Kong Special Administrative Region Research Grant Council under a Direct Grant, Grant CUHK4211/03E, and the Area-of-Excellence in Information Technology.

Proceedings of the Third International Conference on Peer-to-Peer Computing (P2P'03)
0-7695-2023-5/03 $17.00 © 2003 IEEE

in Section 2.1 and review in Section 2.2 some existing works on data reorganization and redundant data update.

2.1 The Server-less Architecture

A server-less video streaming system comprises a pool of fully connected user hosts, henceforth called nodes. Each node has its own CPU, memory and disk storage. Inside each node is a mini video server software that serves a portion of each video title to other nodes in the system. Unlike conventional video server, this mini server software serves a much lower aggregate bandwidth and therefore can readily be implemented in today's STBs and PCs. For large systems, the nodes can be further divided into clusters where each cluster forms an autonomous system that is independent from other clusters.

For data placement, a video title is first divided into fixed-size striping units (or called blocks). Then, these striping units are distributed to all nodes in the cluster in a round-robin manner. This node-level striping scheme avoids data replication while at the same time divides the storage requirement equally among all nodes in the cluster.

To initiate a video streaming session, a receiver node will first locate the set of sender nodes carrying blocks of the desired video title, the striping policy and other parameters (format, bitrate, etc.) through the directory service, which could be provided by a directory server, or peer-to-peer lookup service such as CHORD [9]. These sender nodes will then be notified to start transmitting the video blocks to the receiver node.

Let N be the number of nodes in the cluster and assume all video titles are constant-bit-rate (CBR) encoded at the same bitrate R_v. For a sender node in a cluster, it may have to retrieve video blocks for up to N video streams, of which $N-1$ of them are transmitted while the remaining one played back locally. Note that as a video stream is served by N nodes concurrently, each node only needs to serve a bitrate of R_v/N for each video stream. With a round-based transmission scheduler, a sender node simply transmits one block to each receiver node in each round. Interested readers are referred to the study by Lee and Leung [1] for more details.

2.2 Related Works

The data reorganization problem has been studied in the context of disk arrays [5-6]. The study by Ghandeharizadeh and Kim [5] is the earliest study on data reorganization known to the authors. They investigated the data reorganization problem in the context of adding disks to a continuous media server. They employed round-robin data striping common in disk arrays and investigated and analyzed techniques to perform data reorganization online, i.e., without disrupting on-going video streams.

In another study by Goel et al. [6], a pseudo-random algorithm called SCADDAR for data placement and data reorganization was proposed for use in disk arrays. In this algorithm, each data block is initially randomly distributed to the disks with equal probabilities. When a new disk is added to the disk array, each block will obtain a new sequence number according to their randomized SCADDAR algorithm. If the reminder of this number is equal to the disk number of the newly added disk, the corresponding block will be moved to this new disk. Otherwise, the block will reside at the original disk.

In a recent study [3], Ho and Lee proposed a more efficient data reorganization algorithm called Row-Permutated Data Reorganization that can achieve lower data reorganization overhead and also allow controllable tradeoff between streaming load balance and data reorganization overhead.

While these studies have been successful in reducing the data reorganization overhead substantially, they did not yet address the issue of redundant data update. Given that a server-less video streaming system is built from user hosts that are inherently less reliable than dedicated video servers, fault tolerant capability clearly becomes a necessity. To this end, one will need to incorporate data and capacity redundancies into the system. These redundant data, computed from erasure correction codes such as the Reed-Solomon Erasure Correction (RSE) code, will need to be updated whenever new nodes are added and data reorganization is performed.

The next section explains the redundant data update problem in more details and briefly review the recently proposed Sequential Redundant Data Update (SRDU) algorithm [4].

3. Redundant Data Update

Based on the server-less architecture presented in Section 2.1, let B be the total number of fixed-size video blocks in the system and v_j be the j^{th} block of the video title. For simplicity we consider only one video title although the results can be readily extended to multiple video titles.

Fig. 1 illustrates one possible placement of video blocks in a server-less video streaming system. Each block in the figure represents either a Q-byte video block or a Q-byte redundant data block. Blocks under the same column are stored in the same node with d_i and r_i denoting the data nodes and redundant nodes. Let $(N-h)$ and h be the number of data nodes and redundant nodes in the system respectively. The j^{th} redundant data block, denoted by $c_{i,j}$, are computed from video data stripe i, comprising blocks $\{v_k,\ k=i(N-h),\ i(N-h)+1,\ \ldots,\ (i+1)(N-h)-1\}\}$, using a systematic erasure-correction code such as the

Reed-Solomon Erasure Correction (RSE) code [7-8].

Briefly speaking, with h redundant data blocks in a data stripe, the system will be able to sustain the failure of up to h nodes without loosing any data. A previous study [2] had shown that one can achieve system-level reliability comparable to high-end dedicated video server with redundancies of $h/(N\text{-}h)\approx 0.2$.

When one or more new nodes join the system, they will add both streaming load as well as capacity to the system. Before assimilating them into the system, portion of video blocks must be reorganized among them to utilize the streaming and storage capacity and this is the data reorganization problem discussed before. Consequently, due to the relocation of some video blocks, the redundant data that are computed from the data stripe will require corresponding update and this redundant data update problem will incur overhead in transmitting data blocks to the nodes for regenerating the redundant data blocks. We will first present a trivial solution, called redundant data regeneration, in Section 3.1. Then we review the more efficient SRDU algorithm in Section 3.2.

3.1 Redundant Data Regeneration

For a general systematic erasure-correction code in a system with N nodes and h redundancies, we will need all $(N–h)$ data blocks in a stripe to compute the corresponding h redundant data blocks. As individual data and redundant blocks of a stripe are all stored in different nodes, the data blocks will all need to be transmitted to the redundant nodes (i.e., nodes storing the redundant data blocks) for regenerating the new redundant data blocks.

Therefore for a system with B data blocks, a total of B blocks will need to be transmitted to and received by the redundant node to support redundant data regeneration. Clearly this overhead is very significant and worst.

On the other hand, if a central archive server storing all video blocks is available in the system, then it can simply regenerate the new redundant data blocks locally and send them to the redundant nodes to replace the old redundant data blocks. In this case, the number of blocks sent will be reduced by a factor of $(N–h)$ to $(B/(N–h))$. Nevertheless maintaining this central archive server will incur additional costs, and depending on applications, may not be desirable or even feasible.

3.2 Sequential Redundant Data Update

By considering the generation of a redundant data block from a data stripe, we can observe that in most cases, the reorganized data stripe still comprises many data blocks from the old data stripe before reorganization. For example, in growing a system from N nodes to N+1 nodes, the first data stripe will be reorganized from the composition of $\{v_0, v_1, \ldots, v_{N–h–1}\}$ to $\{v_0, v_1, \ldots, v_{N–h–1}, v_{N–h}\}$, which differs by only one data block $v_{N–h}$. This is the key idea behind SRDU algorithm, which reuses the old redundant block to compute the new redundant block such that only a portion of the data stripe will be needed to transmit.

Among different erasure correction codes there is a class of codes called linear systematic block erasure correction codes, with the Reed-Solomon Erasure Correction (RSE) code being one well-known example. One key property of linear systematic block codes is the use of strictly linear matrix multiplications in computing the redundant data, and this very property enables us to reuse original redundant data to compute the updated redundant data.

Specifically, let $(N\text{-}h)$ and h be the number of data nodes and redundant nodes in the system respectively. Assuming the number of redundant nodes in the system is fixed, then we can apply the (N, h)-RSE code to compute the h redundant data blocks from each stripe of $(N\text{-}h)$ data blocks using

$$
\begin{aligned}
F \cdot D &= \begin{bmatrix} f_{1,1} & f_{1,2} & f_{1,3} & \cdots & f_{1,N-h} \\ f_{2,1} & f_{2,2} & f_{2,3} & \cdots & f_{2,N-h} \\ \vdots & \vdots & \vdots & & \vdots \\ f_{h,1} & f_{h,2} & f_{h,3} & \cdots & f_{h,N-h} \end{bmatrix} \begin{bmatrix} d_{i,0} \\ d_{i,1} \\ \vdots \\ d_{i,N-h-1} \end{bmatrix} \\
&= \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & 2 & 3 & \cdots & N-h \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & 2^{h-1} & 3^{h-1} & \cdots & (N-h)^{h-1} \end{bmatrix} \begin{bmatrix} d_{i,0} \\ d_{i,1} \\ \vdots \\ d_{i,N-h-1} \end{bmatrix} \quad (1) \\
&= \begin{bmatrix} c_{i,0} \\ c_{i,1} \\ \vdots \\ c_{i,h-1} \end{bmatrix} = C
\end{aligned}
$$

where the F, D, and C are the Vandermonde matrix [7], the video data vector, and the redundant data vector respectively; and $d_{i,j}$, $c_{i,k}$ represent data block j (j=0,1,…,N–h–1) and redundant block k (k=0,1,…,h–1) of stripe i respectively. Elements in F is computed from $f_{i,j} = j^{i-1}$ and are constants. Note that the matrix multiplication in (1) is computed over Galois Fields of 2^w where $N < 2^w$. For example, by setting w=16 then the code can support up to 65,535 nodes.

To illustrate the SRDU algorithm, consider the placement in Fig. 1 and Fig. 2, which represent respectively the system configuration before and after the addition of two new nodes. Now, consider the first three old redundant data in redundant node r_1, denoted by $c_{0,1}$, $c_{1,1}$ and $c_{2,1}$, which are computed from

$$c_{0,1} = \sum_{j=0}^{3} f_{2,j+1} v_j \quad (2)$$

$$c_{1,1} = \sum_{j=4}^{7} f_{2,j-4+1} v_j \quad (3)$$

$$c_{2,1} = \sum_{j=8}^{11} f_{2,j-8+1} v_j \quad (4)$$

according to (1).

After two new nodes are added, the system configuration will be changed to that in Fig. 2. Now the two new redundant data, denoted by $c'_{0,1}$ and $c'_{1,1}$, are computed from

$$c'_{0,1} = \sum_{j=0}^{5} f_{2,j+1} v_j \quad (5)$$

and

$$c'_{1,1} = \sum_{j=6}^{11} f_{2,j-6+1} v_j \quad (6)$$

Comparing (5) with (2) we can observe that they share four common terms in v_j: v_0, v_1, v_2, v_3. Thus we can rewrite (5) as follows

$$c'_{0,1} = c_{0,1} + f_{2,5} v_4 + f_{2,6} v_5 \quad (7)$$

That is, by reusing the old redundant data $c_{0,1}$, the overhead decreases to two data blocks instead of six data blocks.

For $c'_{1,1}$, comparing (6) with (4) we can again observe that they share four common terms in v_j: v_8, v_9, v_{10}, v_{11}. However, rewriting (6) cannot be expressed directly using (4) due to the different coefficients $f_{i,j}$ (e.g. $f_{2,1}v_8$ in $c_{2,1}$ versus $f_{2,3}v_8$ in $c'_{1,1}$). To tackle this problem, we can reshuffle the order of computations for $c'_{1,1}$ as

$$\begin{aligned} c'_{1,1} &= \sum_{j=8}^{11} f_{2,j-8+1} v_j + f_{2,5} v_6 + f_{2,6} v_7 \\ &= c_{2,1} + f_{2,5} v_6 + f_{2,6} v_7 \end{aligned} \quad (8)$$

thus enabling us to reuse $c_{2,1}$ in the computation and reducing the number of data block transmissions from six to two.

As observed in (6) and (3), there are also two common terms in v_j, i.e., v_6, v_7, in computing $c_{1,1}$ and $c'_{1,1}$. Again if we reshuffle the parity group order, we can reuse $c_{1,1}$ to construct $c'_{1,1}$, but then the overhead induced will be greater than that of reusing $c_{2,1}$. In this case we simply choose to reuse $c_{2,1}$ instead of $c_{1,1}$. Interested readers are referred to the study by Ho and Lee [4] for more details.

Although the SRDU algorithm can substantially reduce the overhead, the redundancy update overhead is still not insignificant. Intuitively, if we defer the update until more nodes have joined the system, then further savings in update overhead should be achievable.

Let say we defer the update further until four nodes have joined the system as shown in Fig. 3. Now the new redundant data block $c'_{0,1}$ is computed from

$$c'_{0,1} = \sum_{j=0}^{7} f_{2,j+1} v_j \quad (9)$$

$c'_{0,1}$ share four common terms: v_0, v_1, v_2, v_3, with $c_{0,1}$ and thus we can rewrite it as

$$c'_{0,1} = c_{0,1} + \sum_{j=4}^{7} f_{2,j+1} v_j \quad (10)$$

On the other hand, it also share four common terms: v_4, v_5, v_6, v_7, with $c_{1,1}$ and so we can also rewrite it as

$$c'_{0,1} = c_{1,1} + \sum_{j=0}^{3} f_{2,j+4+1} v_j \quad (11)$$

Nevertheless, $c'_{0,1}$ can only be written in terms of one of $c_{0,1}$ or $c_{1,1}$, but not both. Thus although there are eight terms in common, only up to four can be reused. This limitation arises from the original number of data node e.g. four nodes in this case. In particular, as both $c_{0,1}$ and $c_{1,1}$ are computed using the same coefficients $f_{2,1}$ to $f_{2,4}$, this prevents us from reusing both $c_{0,1}$ and $c_{1,1}$ in constructing $c'_{0,1}$. Motivated by this observation, we develop in the next section a new Transpositional Redundant Data Update (TRDU) algorithm that is free from this redundant data reuse limitation.

4. Transpositional Redundant Data Update

In the SRDU algorithm, all the data blocks are inputted to the data vector D in (1) starting from $d_{i,0}$ to $d_{i,N-h-1}$ sequentially. This leads to the limitation in common terms for reusing as explained in Section 3.2. To tackle this shortcoming, we need to increase the number of reusable common terms. The idea is to compute the redundant block using transposed coefficients in the Vandermonde matrix.

Let N_{max} be the maximum size that the system can scale up to, i.e., $N \le N_{max}$. Instead of using (1) for encoding, we replace it with

$$\begin{aligned} F \cdot D &= \begin{bmatrix} f_{1,1} & f_{1,2} & f_{1,3} & \cdots & f_{1,N_{\max}-h} \\ f_{2,1} & f_{2,2} & f_{2,3} & \cdots & f_{2,N_{\max}-h} \\ \vdots & \vdots & \vdots & & \vdots \\ f_{h,1} & f_{h,2} & f_{h,3} & \cdots & f_{h,N_{\max}-h} \end{bmatrix} \begin{bmatrix} d_{i,0} \\ d_{i,1} \\ \vdots \\ d_{i,N_{\max}-h-1} \end{bmatrix} \\ &= \begin{bmatrix} c_{i,0} \\ c_{i,1} \\ \vdots \\ c_{i,h-1} \end{bmatrix} = C \end{aligned} \quad (12)$$

where $d_{i,j}$, $c_{i,k}$ and $f_{i,j}$ have the same definition as (1). The key difference between (1) and (12) is that the encoding matrix in (1) has (N–h) columns, but the same in (12) has (N_{max}–h) columns, which is fixed irrespective of the current system size N.

For each stripe i of data blocks, the starting position of it in the data vector D is shifted by (i(N–h)) mod (N_{max}–h)) and all other $d_{i,j}$ are zeroed. For example, with (N_{max}–h) being 10 and the data placement in Fig. 1, $c_{0,1}$, $c_{1,1}$ and $c_{2,1}$ will be written as

$$c_{0,1} = \sum_{j=0}^{3} f_{2,j+1} v_j \tag{13}$$

$$c_{1,1} = \sum_{j=4}^{7} f_{2,j+1} v_j \tag{14}$$

$$c_{2,1} = \sum_{j=8}^{9} f_{2,j+1} v_j + \sum_{j=10}^{11} f_{2,j-10+1} v_j \tag{15}$$

Note that the sequence of data blocks in computing $c_{2,1}$ spans beyond the boundary of D and in this case, they will be looped back to the beginning. As nearby redundant blocks are computed using different coefficients $f_{i,j}$ in the Vandermonde matrix, we can reuse the old redundant data in updating the new redundant data and thus, eliminating the limitation of the SRDU algorithm.

Although the coefficient $f_{2,1}$ of v_0 in (13) and v_{10} in (15) is the same, this would not affect the update as long as the maximum number of data nodes is fixed at 10, which prevent v_0 and v_{10} from residing in the same data stripe in computing the same redundant data block.

Assume that two data nodes are added to the system as shown in Fig. 2. The first two new redundant data, denoted by $c'_{0,1}$ and $c'_{1,1}$, are computed from

$$c'_{0,1} = \sum_{j=0}^{5} f_{2,j+1} v_j \tag{16}$$

and

$$c'_{1,1} = \sum_{j=6}^{9} f_{2,j+1} v_j + \sum_{j=10}^{11} f_{2,j-10+1} v_j \tag{17}$$

$c'_{0,1}$ can now be constructed by $c_{0,1}$ with the overhead of two data blocks

$$\begin{aligned} c'_{0,1} &= \sum_{j=0}^{3} f_{2,j+1} v_j + f_{2,5} v_4 + f_{2,6} v_5 \\ &= c_{0,1} + f_{2,5} v_4 + f_{2,6} v_5 \end{aligned} \tag{18}$$

and for $c'_{1,1}$, it can be written as

$$\begin{aligned} c'_{1,1} &= \sum_{j=6}^{9} f_{2,j+1} v_j + \sum_{j=10}^{11} f_{2,j-10+1} v_j \\ &= \sum_{j=6}^{7} f_{2,j+1} v_j + \sum_{j=8}^{9} f_{2,j+1} v_j + \sum_{j=10}^{11} f_{2,j-10+1} v_j \\ &= (\sum_{j=4}^{7} f_{2,j+1} v_j - f_{2,5} v_4 - f_{2,6} v_5) + \\ &\qquad (\sum_{j=8}^{9} f_{2,j+1} v_j + \sum_{j=10}^{11} f_{2,j-10+1} v_j) \\ &= (c_{1,1} - f_{2,5} v_4 - f_{2,6} v_5) + c_{2,1} \end{aligned} \tag{19}$$

However, v_4 and v_5 are already transmitted in constructing $c'_{0,1}$ (c.f. Equation (18)) and thus by caching the data blocks, the overhead in constructing $c'_{1,1}$ is in fact zero. An important point to notice is that rewriting $c'_{1,1}$ as (19) is not allowed in the SRDU algorithm (c.f. Equation (8)) because both v_6 and v_{10} share the same coefficient $f_{2,3}$, and both v_7 and v_{11} share the same coefficient $f_{2,4}$. However, by shifting the coefficients as done in TRDU, their coefficients become different and thus allowing more efficient reuse of the old redundant data.

In the extreme case, when the number of additional data node equals to integral multiples of the original system size, the overhead will become zero as the redundant node can compute the new redundant data blocks simply by combining old redundant data blocks locally. For example, assume four data nodes are entered, with data placement shown in Fig. 3. Now, the new redundant data block $c'_{0,1}$ will be equal to

$$c'_{0,1} = \sum_{j=0}^{7} f_{2,j+1} v_j \tag{20}$$

which can be directly written as

$$\begin{aligned} c'_{0,1} &= \sum_{j=0}^{3} f_{2,j+1} v_j + \sum_{j=4}^{7} f_{2,j+1} v_j \\ &= c_{0,1} + c_{1,1} \end{aligned} \tag{21}$$

It does not involve any data blocks at all, and so the overhead is zero.

Compared to SRDU, there are also tradeoffs in using TRDU. First, we need to specify the maximum system size N_{max} in advance. To avoid hitting the limit, we may need to use a large value for N_{max}. This creates the second tradeoff, where the larger Vandermonde matrix will increase the computational complexity in decoding as we need to compute the inverse of the larger matrix. Finally, due to the larger matrix size, TRDU will also increase the memory consumed in performing the matrix operations during decoding.

5. Multiple Redundant Nodes

So far only the overhead for updating one redundant node is considered. In this section, we extend all the presented algorithms to systems with multiple redundant nodes.

First, we consider the redundant data regeneration algorithm (c.f. Section 3.1). Fig. 4 illustrates the redundant data regeneration process. Note that we need to transmit all data blocks to the redundant node r_0 to compute the new redundant data blocks. For a video title of B data blocks, this step will incur an overhead of B blocks. Additionally, after r_0 regenerates all the updated redundant data blocks, it will need to transmit the updated redundant blocks to the other redundant nodes as well and this incurs another overhead of $(B/(N–h))$ blocks for each additional redundant node. Therefore, the total redundant data update overhead is equal to $B+(h–1)(B/(N–h))$.

Second, if there is a central archive server storing a copy of all the video blocks, it can regenerate locally all the updated redundant data blocks and then transmit them to the redundant nodes (Fig. 5). The overhead in this case will be equal to $h(B/(N–h))$.

Third, Fig. 6 illustrates the process in SRDU and TRDU. The data blocks needed for computing the updated redundant data are first transmitted to the redundant node r_0. Then, partial results are computed in r_0 and transmitted to other redundant nodes for computing their updated redundant data blocks. For example, assume two new nodes are added and consider the first redundant block $c'_{0,j}$ in the redundant node r_j. The equation for computing $c'_{0,j}$ (c.f. Equation (7) and (18)) is

$$c'_{0,j} = c_{0,j} + f_{j+1,5}v_4 + f_{j+1,6}v_5 \qquad (22)$$

As v_4 and v_5 are already transmitted to r_0 according to the SRDU/TRDU algorithm, r_0 can then compute the partial result $(f_{j+1,5}v_4 + f_{j+1,6}v_5)$ for all $j \neq 0$. In the redundant node r_j, this partial result can be combined with the original redundant block $c_{0,j}$ to generate the new redundant block $c'_{0,j}$. The transmission of partial result will introduce an overhead of $(B/(N–h))$ blocks for each additional redundant node. As a result, the total overhead is equal to the block movement overhead under SRDU/TRDU plus the overhead in transmitting the partial results $(h\text{-}1)(B/(N–h))$.

Table 1 summarizes the total overhead of the redundant data update algorithms studied. We can observe that the overhead is dominated by the overhead in updating the first redundant node because of the large number of data blocks needed to generate the new redundant data blocks. Once these are cached in the redundant node r_0, new redundant blocks of other redundant nodes can be computed with much lower overhead.

Table 1. Total overhead in studied algorithms.

Algorithms	Total Overhead
Redundant Data Regeneration	$B+(h–1)(B/(N–h))$
Regeneration by archive server	$B/(N–h)+$ $(h–1)(B/(N–h))$
Sequential Redundant Data Update (SRDU)	Block movement under SRDU+ $(h–1)(B/(N–h))$
Transpositional Redundant Data Update (TRDU)	Block movement under TRDU+ $(h–1)(B/(N–h))$

6. Performance Evaluation

In this section, we evaluate the studied algorithms using numerical results. As the overhead is dominated by updating the first redundant node, for simplicity we will ignore the overhead in updating additional redundant nodes. The overhead in updating additional redundant nodes can be easily obtained according to the analysis in Section 5.

Beginning with a small system, we add new nodes to the system and then apply the studied algorithms to update the redundant data blocks. Performance is measured by the number of data blocks that need to be sent to the redundant nodes – or simply called redundancy update overhead. The total number of data blocks is 40,000 and is fixed throughout the system lifetime.

6.1 Continuous System Growth

In the first experiment, we begin with a system of five data nodes and one redundant node. Then we add a new node to the system one by one, each time the redundant data blocks are completely updated using different algorithms. This continues until the system grows to 400 data nodes.

Fig. 7 plots the redundancy update overhead versus system size from 6 to 400. As expected, Redundant Data Regeneration performs the worst, essentially requiring all data blocks to be sent to the redundant node for regenerating the redundant data. On the other hand, regenerating redundant data using a central archive server incurs the least overhead, albeit at the expense of extra central facility. SRDU and TRDU perform similarly, with TRDU achieving slightly less overhead.

6.2 Batched Update

In the previous experiment, we always completely update all redundant data blocks before adding another

new node. Clearly this is inefficient if new nodes are added frequently or added to the system in a batch. To address this issue, we conduct a second experiment where redundant data blocks are not updated until a fixed number of nodes, say *W*, are added – *batched redundancy update*. During this time, storage and streaming capacity in the new nodes are not utilized and thus this approach represents tradeoffs between redundancy update overhead and resource utilization. Fig. 8 plots the redundancy update overhead versus the batch size *W* for initial system size of 80 data nodes. The key observation is that the normalized per-node redundancy update overhead of both the SRDU and TRDU algorithms decreases significantly with the batch size.

Moreover, TRDU performs significantly better than SRDU when the batch size is large. This is because the reusing of old redundant data in SRDU is limited by the original number of data nodes, while that in TRDU will increase with the batch size.

Furthermore, as discussed in section 4, the overhead of TRDU is equal to zero when the batch size is integer multiples of the initial data node size. For example, in Fig. 8 the two zero overhead points occur when the batch size is 80 and 160.

7. Conclusion and Future Works

This study shows that we can reduce the redundancy update overhead significantly by combining the use of batched update and the Transpositional Redundant Data Update algorithm. It also reveals that the additional redundancy overhead for multiple redundant nodes is insignificant, thereby paving the way for employing multiple redundant nodes to increase the reliability of server-less video streaming systems.

Nevertheless, there are still many open problems in growing a server-less video streaming system. For example, when the system grows larger with more nodes, the system reliability will decrease unless additional redundant nodes are added to compensate. However, due to the orthogonal nature of the redundant data, the new redundant data cannot be computed from the existing redundant data and so must be generated directly from the data blocks. On the other hand, nodes in the system are only peers, which may readily leave the system anytime. The shrinking of the system may introduce several problems, including redundant data update, data reorganization and fault tolerance etc. Therefore further investigation is warranted to address these challenges to build a truly scalable server-less video streaming system.

References

[1] Jack Y. B. Lee and W. T. Leung, "Study of a Server-less Architecture for Video-on-Demand Applications," *Proc. IEEE International Conference on Multimedia and Expo*., August 2002.

[2] Jack Y. B. Lee and W. T. Leung, "Design and Analysis of a Fault-Tolerant Mechanism for a Server-less Video-on-Demand System," *Proc. 2002 International Conference on Parallel and Distributed Systems*, Taiwan, Dec 17-20, 2002.

[3] T. K. Ho and Jack Y. B. Lee, "A Row-Permutated Data Reorganization Algorithm for Growing Server-less Video-on-Demand Systems," *Proc. International Symposium on Cluster Computing and the Grid 2003*, Tokyo, Japan.

[4] T. K. Ho and Jack Y. B. Lee, "A Novel Redundant Data Update Algorithm for Fault-Tolerant Server-less Video-on-Demand Systems," *Proc. 2003 High Performance & Large Scale Computing (HP&LSC) Conference*, Nottingham, UK, June 9-11, 2003.

[5] S. Ghandeharizadeh and D. Kim, "On-line Reorganization of Data in Scalable Continuous Media Servers," *Proc. 7th International Conference on Database and Expert Systems Applications*, September 1996.

[6] A. Goel, C. Shahabi, S.-Y. Yao, and R. Zimmerman, "SCADDAR: An Efficient Randomized Technique to Reorganize Continuous Media Blocks," *Proc. International Conference on Data Engineering*, 2002.

[7] J. S. Plank, "A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems," *Software Practice and Experience*, vol.27(9), Sep. 1997, pp.995-1012.

[8] L. Rizzo, "Effective Erasure Codes for Reliable Computer Communication Protocols," *ACM Computer Communication Review*, vol.27(2), Apr 1997, pp.24-36.

[9] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications", *ACM SIGCOMM 2001*, San Deigo, CA, August 2001.

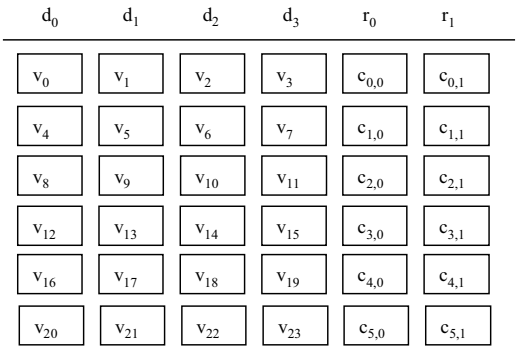

Fig. 1. Original data placement before addition of nodes.

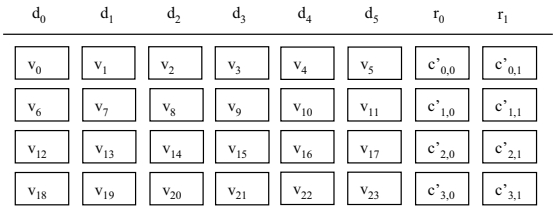

Fig. 2. Data placement after adding two data nodes.

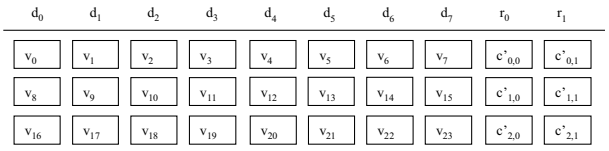

Fig. 3. Data placement after adding four data nodes.

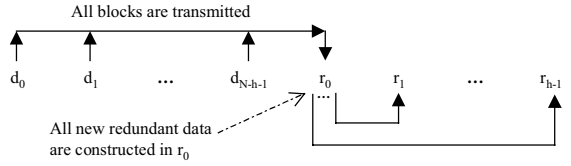

Fig. 4. Redundant data regeneration when video blocks are fully distributed.

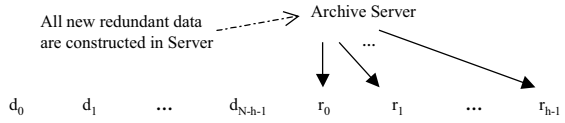

Fig. 5. Redundant data regeneration using an archive server.

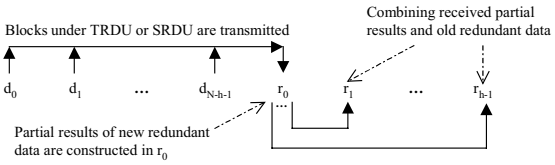

Fig. 6. Redundant data update in SRDU/TRDU.

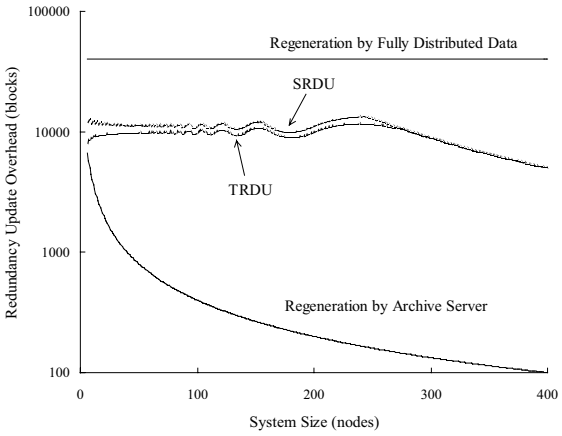

Fig. 7. Redundancy update overhead versus system size.

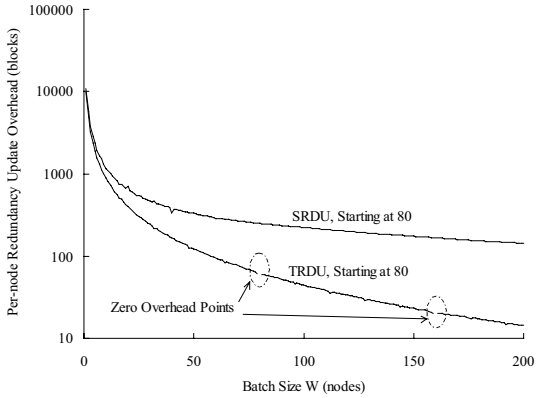

Fig. 8. Per-node redundancy update overhead versus batch size.

Proceedings of the Third International Conference on Peer-to-Peer Computing (P2P'03)
0-7695-2023-5/03 $17.00 © 2003 **IEEE**