

# New Protocols for Remote File Synchronization Based on Erasure Codes\*

Utku Irmak Svilen Mihaylov Torsten Suel

CIS Department  
Polytechnic University  
Brooklyn, NY 11201

uirmak@cis.poly.edu, smihay01@utopia.poly.edu, suel@poly.edu

## Abstract

Given two versions of a file, a current version located on one machine and an outdated version known only to another machine, the remote file synchronization problem is how to update the outdated version over a network with a minimal amount of communication. In particular, when the versions are very similar, the total data transmitted should be significantly smaller than the file size. In this paper, we present a new approach to file synchronization based on the use of erasure codes. Using this approach, we design a single-round protocol that is provably efficient with respect to common measures of file distance, and another optimized practical protocol that shows promising improvements over *rsync* on our data sets.

## 1 Introduction

Consider the problem of maintaining replicated collections of files, such as user files, web pages, or documents, over a slow network. In particular, assume that we have two machines,  $A$  and  $B$ , that each hold a copy of the files, and that files may have been updated at one of the machines. Periodically, a machine may initiate a synchronization to update all its replicas to the latest version. If the file or collection is large or the network slow, then it is desirable to perform this synchronization with a minimum amount of communication over the network.

The above scenario arises in a number of applications. The most common ones include: (1) Synchronization of user files: Both the *rsync* [19, 21] and *unison* [13] tools are widely used to synchronize personal files between different machines which may only be connected over a slow network such as a modem. (2) Web and ftp site mirroring: *rsync* is widely used to mirror busy web and ftp sites, including sites distributing new versions of software. In this case, there may be significant similarities between successive versions of a software package that allow a mirror to efficiently update to the newest release. (3) Content distribution networks: Several companies in the CDN space have studied and deployed file synchronization techniques sim-

ilar to *rsync*. We are not aware of any published work in this direction, but file synchronization techniques are a natural approach for updating content replicated at the network edge. (4) Web access over slow links: A user revisiting a web page may already have a previous version of the page in the browser cache, and it would be desirable to avoid transmission of the entire updated version. This idea is, e.g., implemented in the *rproxy* system [20], which uses the *rsync* algorithm to update pages that are being revisited.

**Problem Formalization:** In this paper, we focus on this problem of updating files in a bandwidth efficient manner; we refer to this as the *remote file synchronization problem*. We note that a very widely used open source software tool called *rsync*, included in many Linux distributions, addresses this problem based on the *rsync* algorithm in [19, 21].

The setup for the file synchronization problem is as follows. We have two files (strings)  $f_{new}, f_{old} \in \Sigma^*$  over some alphabet  $\Sigma$  (most methods are character/byte oriented), and two machines  $C$  (the client) and  $S$  (the server) connected by a communication link. We also refer to  $f_{old}$  as the *outdated file* and to  $f_{new}$  as the *current file*. We assume that  $C$  only has a copy of  $f_{old}$  and  $S$  only has a copy of  $f_{new}$ . Our goal is to design a protocol between the two parties that results in  $C$  holding a copy of  $f_{new}$  while minimizing the communication cost, i.e., the total number of bits exchanged between the two parties.

The communication cost incurred by the protocol should depend on the degree of similarity between the two files. Similarity is usually defined in terms of one of a number of edit distance measures that have been proposed, see [3, 2, 4] for example. We focus mainly on the *edit distance with block moves*, which seems powerful enough to be used as a reasonable model of file similarity, but still simple enough to work with: The edit distance with block moves is the smallest number of insertions, deletions, and changes of single symbols or moves of blocks of symbols needed to transform one file into the other. For technical reasons, we assume that each block move operation adds 3 to the distance, while other operations add 1.

We limit ourselves to a single round of messages between client and server, since multiple rounds are undesirable in many scenarios involving small files or large latencies, such as retrieving a single HTML page over a high-latency modem con-

---

\*Work supported by NSF CAREER Award CCR-0093400, NSF ITR Award CNS-0325777, and the Wireless Internet Center for Advanced Technology (WICAT) at Polytechnic University.

nection. In addition, single-round protocols can be more easily integrated into existing tools currently relying on *rsync*, and multi-round protocols can introduce other complications due to state that may have to be kept at the server for best performance [8]. Multi-round protocols are suitable when dealing with large files, or with large collections of files since the multiple communication rounds are not incurred on a per-file basis but can be overlapped for different files.

Before continuing, we point out a few assumptions. We assume that collections consist of unstructured files that may be modified in arbitrary ways, including insertion and deletion operations that change line and page alignments between different versions. Thus, approaches that identify changed disk pages or bit positions or that assume fixed record boundaries do not work. We note that the problem would also be easier if all update operations to the files are saved in an update log that can be transmitted to the other machine, or if the machine holding the current version has a copy of the outdated version. However, in many scenarios this is not the case. We are not concerned with issues of consistency in between synchronization steps, and with the question of how to resolve conflicts if changes are simultaneously performed at several locations [1, 14]. We assume a simple two-party scenario where it is known which files need to be updated and which is the current version.

**Contributions of this Paper:** We describe a new approach to single-round file synchronization based on the use of erasure codes. Using this approach, we derive a protocol that communicates at most  $O(k \lg(n) \lg(n/k))$  bits on files with edit distance with block moves of at most  $k$ . To our knowledge this is the first single-round protocol that is both feasible and communication-efficient. Also, using the same approach, we derive another algorithm and an optimized implementation that achieves very promising improvements over *rsync* on a range of test data. The results are still preliminary and we expect additional improvements in the final version of this paper.

## 2 Related Work

The *rsync* algorithm [19, 21] is the basis of the very widely used *rsync* open source tool, which clearly provides a useful improvement over the alternative of transmitting the entire file. However, *rsync* does not guarantee any strong performance bounds with respect to common file distance measures.

There are a number of theoretical studies of the file synchronization problem [3, 2, 9, 10]. In particular, Orlitsky [9, 10] presents almost tight bounds for the problem with varying numbers of communication phases, under some assumptions about the assumed file distance metric. Since they assume that the receiver can invert a hash function over a large domain in order to decode the current version of the file, these results typically require exponential time for decoding; while this is allowable under the standard model for communication complexity [6], it makes the algorithms impractical. Within this framework, [12] discusses a relationship between Error Correcting Codes and file synchronization.

Various practical multi-round algorithms are proposed in [16, 3, 2, 4, 11, 7, 17, 8]. These algorithms are based on recursive partitioning of unmatched blocks, mostly in a breadth-first manner with the exception of [4]. Experimental results for multi-round algorithms are provided in [7, 11, 17, 8].

## 3 An Approach Based on Erasure Codes

In this section, we design two algorithms, one primarily of theoretical interest and another one that performs well in practice. The basic idea underlying the new approach is quite simple: essentially, erasure codes are used to convert certain multi-round protocols into single-round protocols with similar communication cost. We start by describing a simple multi-round protocol. Subsection 3.2 contains the theoretical result obtained by converting the multi-round protocol, and Subsection 3.3 describes and evaluates the practical protocol.

### 3.1 A Simple Multi-Round Protocol

We now describe a simple multi-round protocol for file synchronization, which we refer to as the *basic multi-round protocol*. The protocol is not new and variations of it have previously appeared in [16, 3, 11, 17].

The protocol runs in a number of rounds, starting with a block size of  $b_{max}$  and then decreasing the block size by a factor of 2 in each round until reaching a block size of  $b_{min}$ . In the first round, the server holding the current version partitions the file into blocks of size  $b_{max}$ , and sends a hash value for each block to the client. The client attempts to match the received hashes to all possible alignments in the outdated file, and then responds with a bit vector containing a “1” for each hash that found a match, and a “0” for all other hashes. By doing so, the client notifies the server which of the hashes were “understood” by the client, and which hashes could not be decoded by looking for a match in its file.

Next the server partitions each block whose hash did not find a match into two halves, and sends hashes for these smaller blocks to the client. The client again replies with a bit vector, and the server further splits any unmatched blocks. Once block size  $b_{min}$  is reached, the server sends all unmatched blocks.

Suppose we choose  $b_{max} = \lfloor n/k \rfloor_2$ ,  $b_{min} = \lg(n)$ , and use hashes of size, say,  $4 \lg n$  bits (where  $\lfloor n/k \rfloor_2$  denotes the next smaller power of 2 of  $n/k$ , and  $\lg$  is the logarithm with base 2). Then it can be shown that given two files with edit distance with block moves of  $k$ , the algorithm transmits at most  $O(k \lg(n) \lg(n/k))$  bits and correctly updates the file with probability at least  $1 - \frac{1}{n}$ . In particular, on the first level we have at most  $2k$  blocks for which hashes are sent. There are at most  $\lg(n/k)$  levels. On each level at most  $k$  of the hashes that are sent do not find a match at the client, and thus again at most  $2k$  hash values are sent at the child level, as implied by the following simple lemma.

**Lemma 3.1** *Let  $f_{new}$  and  $f_{old}$  be two files with edit distance with block moves at most  $k$ , where each move operation is*

counted as a distance of 3. If we partition  $f_{new}$  into some number  $m$  of blocks  $s_0$  to  $s_{m-1}$ , then at most  $k$  of these blocks do not occur in  $f_{old}$ .

We only sketch the proof of the lemma, which is not really new. Consider a sequence of  $k$  edit operations that transforms  $f_{old}$  into  $f_{new}$ . Imagine that  $f_{old}$  is printed on a long piece of paper, and that each edit operation may require us to cut the piece of paper in order to insert, delete, or change a character at a particular position, or to move a block from one position to another. Each single-character operation requires at most one cut, increasing the number of pieces of the old file by at most one, while each move operation may require up to three cuts. Any substring  $s_i$  that is completely within one of the at most  $k$  pieces clearly also occurs in  $f_{old}$ , giving the result.

Several practical optimizations to this algorithm are described in [17]. Next, we show how to convert this algorithm into a single-round protocol with the same complexity.

### 3.2 An Efficient Single-Round Protocol

First, we define the *complete multi-round algorithm* as the variation of the basic multi-round algorithm where in each round, we split all blocks in half and send hashes for all the resulting smaller blocks, including those whose ancestors have already found matches on a higher level. (Obviously, this is not a communication-efficient algorithm.) Due to the above lemma, both variations have the property that on each level at most  $k$  hashes do not find a match.

Our second required ingredient is a *systematic erasure code*, which we now discuss briefly. We refer to [15] for a more detailed discussion. In an *erasure code*, we are given  $m$  source data items of some fixed size  $s$  each, which are encoded into  $m' > m$  encoded data items of the same size  $s$ , such that if any  $m' - m$  of the encoded data items are lost during transmission, they can be recovered from the at least  $m$  correctly received encoded data items. Note that it is assumed here that a receiver knows which items have been correctly received and which are lost. A *systematic* erasure code is one where the encoded data items consist of the  $m$  source data items plus  $m' - m$  additional items. In our application, which requires a systematic erasure code, the data items are hashes, and we refer to the  $m' - m$  additional items as *erasure hashes*.

Our algorithm is essentially a communication-efficient single-round simulation of the complete multi-round algorithm. Suppose we know an upper bound  $k$  on the edit distance with block moves between the files. Then on each level, we can simulate the complete multi-round algorithm according to the following rules:

- Any hash value sent in the complete multi-round algorithm that would not be sent in the basic multi-round algorithm (since it corresponds to a block whose ancestor has already found a match) is not transmitted, as it can be recreated at the client by evaluating the hash function on the corresponding part of the match.

- Any hash value that would be sent by the basic multi-round algorithm (since it corresponds to a block with no matched ancestors) is also not sent to the client, but considered *lost*.
- Since on each level there can be at most  $2k$  such blocks that are declared *lost*, we can recreate the entire level of hashes at the client by sending  $2k$  extra erasure hashes, computed with a systematic erasure code, and then recovering the *lost* hashes.

To summarize, the algorithm works as follows:

- (1) The server partitions  $f_{new}$  recursively into blocks from size  $b_{max}$  down to  $b_{min}$ , and for each level computes all block hashes.
- (2) The server applies a systematic erasure code to each level of hashes except the top level, and computes  $2k$  erasure hashes for each level.
- (3) In one message, the servers sends all hashes at the highest level to the client, plus  $2k$  erasure hashes for each level.
- (4) The client, upon receiving the message, recovers the hashes on all levels in a top-down manner, by first matching the top-level hashes. Then on the next level, the hash function is applied to all blocks with an ancestor that was matched on a higher level, to compute their hashes. Then the  $2k$  erasure hashes are used to recover the hashes of the at most  $2k$  blocks with no matched ancestors.
- (5) At the bottom level with block size  $b_{min}$ , we assume that the hash is simply the content of the block, and thus we can recover the current file at the client.

Assuming no hash collisions, the algorithm correctly simulates the complete multi-round algorithm. Choosing as before  $b_{max} = \lfloor n/k \rfloor_2$ ,  $b_{min} = \lg(n)$ , and hashes of size  $4 \lg n$  bits we get the following result:

**Theorem 1** *Given a bound  $k$  on the edit distance between  $f_{old}$  and  $f_{new}$ , the erasure-based file synchronization algorithm correctly updates  $f_{old}$  to  $f_{new}$  with probability at least  $1 - \frac{1}{n}$ , using a single message of  $O(k \lg(n) \lg(n/k))$  bits.*

We note that there are highly efficient single-message protocols for estimating file distances according to a variety of edit distance measures; see [3]. These results imply that the above bound can be achieved by a single-round protocol even if there is no a-priori known bound on the file distance  $k$ , if the request message from client to server is used to estimate  $k$ . To our knowledge, this result is the first feasible single-round protocol for file synchronization that is provably communication-efficient with respect to edit distance with block moves, or any distance measure allowing for block operations. Another interesting property of the protocol is that by broadcasting a single message, the current version can be communicated to clients holding different outdated versions.

### 3.3 A Practical Protocol Based on Erasure Codes

While the protocol from the previous subsection is efficiently implementable and has reasonable performance, it does suffer from two main shortcomings.

- The protocol requires us to estimate an upper bound on the file distance  $k$ . Although there are efficient protocols for this, an underestimation would make the recovery impossible at the client, thus to be sure we may have to send more than needed.
- More importantly, the algorithm does not support compression of unmatched literals. The performance of *rsync* and other protocols such as [17] is significantly improved through the use of compression for literals. There are some tricks that one can use to integrate compression but none of them completely resolves the issue.

To address these problems we now design another erasure-based algorithm that works better in practice. The main change is that now, as in *rsync*, hashes are sent from client to server as part of the request, while the server uses the hashes to identify common blocks and then sends the unmatched literals in compressed form. In the algorithm, the first three steps are identical to the previous one but with the roles of client and server exchanged and with  $m_i$  instead of  $2k$  erasures per level.

- (4) The server, upon receiving the message, attempts to recover the hashes on all levels in a top-down manner, by first matching the top-level hashes. Then on the next level  $i$ , if the number of blocks without any matched ancestor is at most  $m_i$ , the hash function is applied to all blocks that do have a matched ancestor, and the  $m_i$  erasure hashes are used to recover the hashes of the other blocks. Otherwise, we stop at the previous level of hashes.
- (5) We now use the hashes on the lowest level that was successfully decoded, in exactly the same way they are used in *rsync*. Thus, common blocks are identified and all unmatched literals are sent in compressed form to the client. For further performance improvements, we apply some basic optimizations over the *rsync* approach which we discuss briefly in Subsection 3.4.

If we set parameters as before, we can show that this algorithm achieves the same performance bounds, assuming an upper bound on the file distance  $k$  that can be used to choose appropriate  $m_i$  (and assuming that compression does not increase the size of the literals). However, even if we do not have an upper bound on  $k$ , the algorithm degrades more gracefully: While the previous algorithm fails to transmit  $f_{new}$  if not enough erasure hashes are available, this algorithm, like *rsync*, will still correctly transmit  $f_{new}$  though possibly at increased cost. In the worst case, when not enough erasure hashes are available to encode any of the lower levels, the algorithm will achieve the same performance as *rsync* on block size  $b_{max}$ .

In practice, we will usually choose  $b_{max}$  to be similar to or slightly larger than the default block size of 700 used by

*rsync*, and then use a smaller value of  $b_{min}$  maybe around 100 to 200 bytes. The  $m_i$  for the different levels are determined as a fraction  $r_i$  of the total number of hashes on a particular block. By assuming some minimum rate of matches on the higher levels we can decrease the cost of hashes at the lower levels and hence afford to go to smaller block sizes on very similar files. We have experimented with a number of choices of  $b_{max}$ ,  $b_{min}$ , and the  $r_i$ .

### 3.4 Implementation Overview and Preliminary Results

In our implementation, we included two additional optimizations over *rsync*. In the first one, we replace the *gzip* algorithm used for the transmission of the unmatched literals and match tokens with an optimized delta compressor. The server now creates a reference file from the contents of all matched blocks, then compresses the current file with respect to this reference file, and transmits the resulting delta. Also, the server sends a (possibly compressed) bit vector telling the client which of its hash values has found a match, allowing the client to create the same reference file and then decode the current file. This has two advantages: First, it exploits redundancies between unmatched and matched parts of the current file [19]. Second, an optimized delta compressor may provide a more efficient way to encode offsets and indices than the tokens in *rsync*. We used the highly optimized *zdelta* delta compressor [18], available at <http://cis.poly.edu/zdelta/>.

In the second optimization, we make a better choice of the number of bits per hash: We assume some upper bound on the probability of a collision, say  $1/2^d$  for some  $d$ , and then use  $\lg(n) + \lg(y) + d$  bits per hash where  $n$  is the file size and  $y$  is the total number of hashes sent from client to server.

We implemented the algorithm using an implementation of systematic erasure codes by Rizzo, available at <http://info.iet.unipi.it/~luigi/fec.html>. The erasure code implementation is based on Vandermonde matrices, and achieves encoding and decoding rates of several MB per second. Given that the hashes are much smaller than the actual files, this translates to a file processing speed of tens to hundreds of MB/s. Thus, we do not expect coding to be a bottleneck. We chose the number of bits per hash as defined above, for  $k = 10$  and  $y$  equal to the total number of hashes (top-level and erasure hashes) sent. We also integrated decomposable hashes into our implementation. This technique, first proposed in [16] and recently rediscovered in [17], allows the hash of a child block to be computed from the hashes of its parent and its sibling, halving the number of hashes transmitted on all except the top level. (Thus, we apply erasure coding only to left siblings and compute the right siblings at the server after decoding left siblings.)

For the experiments, we used the *gcc* and *emacs* data sets also used in [5, 17], consisting of versions 2.7.0 and 2.7.1 of *gcc* and 19.28 and 19.29 of *emacs*. In Figure 3.1 we show preliminary experimental results of the cost of updating all files

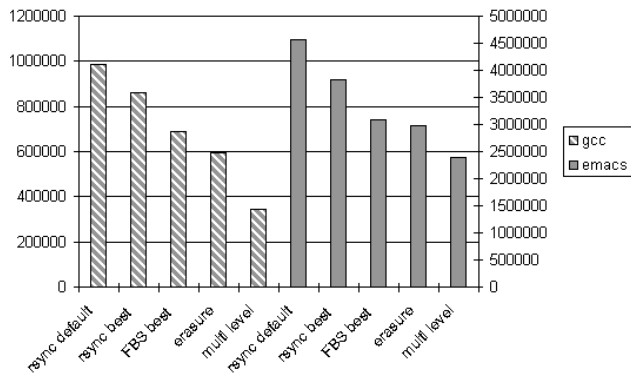


Figure 3.1. Results for the best possible settings.

in the older version to the newer one, comparing the *rsync* approach, our erasure-based algorithm, and the highly optimized multi-round algorithm from [17]. For *rsync* we show three results, one for the default settings, one for the best choice of block sizes, and one for a modified version (labeled FBS) that uses a delta compressor and shorter hashes as described above. For each method, we show the best result that we obtained. The result from [17] essentially provides a limit on what we can realistically hope to gain with a single-round approach, barring further breakthroughs in techniques.

## 4 Conclusions and Open Questions

In this paper, we have described a new approach to remote file synchronization based on the use of erasure codes. Using this approach, we have derived a single-round protocol that is feasible and communication-efficient with respect to a common file distance measure, and another protocol that shows promising improvements over *rsync* in experiments. We expect additional gains once we fully optimize the implementation and parameter settings. We plan to make a stable and high-performance version of the new practical algorithm available in the near future, as part of a library of file synchronization operations. We expect that our approach can be used to derive other interesting single- and multi-round protocols. In addition, there are a number of interesting theoretical open questions on file synchronization problems. The current communication bounds for feasible protocols are still a logarithmic factor from the lower bounds for most interesting distance metrics, even for multi-round protocols. For distance metrics allowing block copies and deletions, the gap is even larger and the best bounds [2, 3, 4] either require a very large number of communication rounds or are more than a logarithmic factor away from optimal in terms of bandwidth use.

## References

[1] S. Balasubramaniam and B. Pierce. What is a file synchronizer? In *Proc. of the ACM/IEEE MOBICOM'98 Conference*, pages 98–108, Oct. 1998.

[2] G. Cormode. *Sequence Distance Embeddings*. PhD thesis, University of Warwick, January 2003.

[3] G. Cormode, M. Paterson, S. Sahinalp, and U. Vishkin. Communication complexity of document exchange. In *Proc. of the ACM-SIAM Symp. on Discrete Algorithms*, Jan. 2000.

[4] A. Evmimievski. A probabilistic algorithm for updating files over a communication link. In *Proc. of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 300–305, Jan. 1998.

[5] J. Hunt, K.-P. Vo, and W. Tichy. Delta algorithms: An empirical analysis. *ACM Transactions on Software Engineering and Methodology*, 7, 1998.

[6] E. Kushilevitz and N. Nisan. *Communication Complexity*. Cambridge University Press, 1997.

[7] J. Langford. Multiround *rsync*. January 2001. Unpublished manuscript.

[8] P. Noel. An efficient algorithm for file synchronization. Master's thesis, Polytechnic University, 2004.

[9] A. Orlitsky. Worst-case interactive communication II: Two messages are not optimal. *IEEE Transactions on Information Theory*, 37(4):995–1005, July 1991.

[10] A. Orlitsky. Interactive communication of balanced distributions and of correlated files. *SIAM Journal of Discrete Math*, 6(4):548–564, 1993.

[11] A. Orlitsky and K. Viswanathan. Practical algorithms for interactive communication. In *IEEE Int. Symp. on Information Theory*, June 2001.

[12] A. Orlitsky and K. Viswanathan. One-way communication and error-correcting codes. In *Proc. of the 2002 IEEE Int. Symp. on Information Theory*, page 394, June 2002.

[13] B. Pierce. Unison file synchronizer. <http://www.cis.upenn.edu/~bcpierce/unison/>.

[14] N. Ramsey and E. Csirmaz. An algebraic approach to file synchronization. In *Proc. of the 9th ACM Int. Symp. on Foundations of Software Engineering*, pages 175–185, 2001.

[15] L. Rizzo. Effective erasure codes for reliable computer communication protocols. *Computer Communication Review*, April 1997.

[16] T. Schwarz, R. Bowdidge, and W. Burkhard. Low cost comparison of file copies. In *Proc. of the 10th Int. Conf. on Distributed Computing Systems*, pages 196–202, 1990.

[17] T. Suel, P. Noel, and D. Trendafilov. Improved file synchronization techniques for maintaining large replicated collections over slow networks. In *Proc. of the Int. Conf. on Data Engineering*, March 2004.

[18] D. Trendafilov, N. Memon, and T. Suel. *zdelta: a simple delta compression tool*. Technical Report TR-CIS-2002-02, Polytechnic University, CIS Department, June 2002.

[19] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, April 2000.

[20] A. Tridgell, P. Barker, and P. MacKerras. *rsync in http*. In *Conference of Australian Linux Users*, 1999.

[21] A. Tridgell and P. MacKerras. The *rsync* algorithm. Technical Report TR-CS-96-05, Australian National University, June 1996.