Improved Single-Round Protocols for Remote File Synchronization

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Abstract— **Given two versions of a file, a current version located on one machine and an outdated version known only to another machine, the remote file synchronization problem is how to update the outdated version over a network with a minimal amount of communication.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. File synchronization problems arise in many application scenarios such as web site mirroring, file system backup and replication, and web access over slow links. An open source tool for this problem, called *rsync* and included in many Linux distributions, is widely used in such scenarios. *rsync* uses a single round of messages between the two machines. While recent research has shown that significant additional savings in bandwidth consumption are possible through the use of optimized multi-round protocols, there are many scenarios where multiple rounds are undesirable.**

In this paper, we study single-round protocols for file synchronization that offer significant improvements over *rsync*. Our main contribution is 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. In addition, we show how to obtain moderate improvements by engineering the *rsync* approach.

I. 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 operation that updates all its replicas to the latest version. This operation involves identifying all files that have changed, deciding which version of the file is the latest one (if files can be changed at either location), and finally updating any outdated files. Or alternatively, when a particular file is accessed, a machine may have to update its local version of the file. If the file or file collection is large or the network fairly 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, such as synchronization of user files between different machines, distributed file systems, remote backups, mirroring of large web and ftp sites, content distribution networks, or web access, to name just a few. In many cases, updated files differ only slightly from their previous version; for example, updated web pages usually change only in a few places. In this case, instead of sending the entire updated version over the network, it would be desirable to perform the update by sending only an amount of data proportional to the degree of change between the two versions.

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 there is a very widely used open source software tool called *rsync*, included in many Linux distributions, that addresses this problem and that is based on the *rsync* algorithm and protocol described in [39], [41]. Another popular tool called *unison* [28] also uses the same basic algorithm. Our goal is to derive new algorithms that achieve significant savings over the *rsync* algorithm in the case of slow networks. We focus on approaches that exchange only a single round of messages between the machines holding the outdated and current version of a file; such approaches are preferable in a number of scenarios as discussed later.

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 – though some of them are potentially useful for identifying those files that have been changed and need to be synchronized. 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 (see [3], [29] for a discussion). We assume a simple two-party scenario where it is known which files need to be updated and which is the current version of a file.

A. Applications

We now discuss the most common application scenarios for file synchronization techniques.

- **Synchronization of user files:** Both the *rsync* and *unison* tools are widely used to synchronize personal files between different machines, say between a machine at home and one at work, that may only be connected over a slow network such as a modem.
- **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.
- **Content distribution networks:** Several companies in the CDN space have studied and deployed file synchronization techniques similar 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 or at several location of a company intranet.
- **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 [40], which uses the *rsync* algorithm to efficiently update pages that are being revisited.

In addition, there are several other scenarios where the techniques could be employed, such as replication of content in a P2P or grid environment, sharing of large web page archives for mining and web search, distributed backup, or wide-area distributed file systems.

B. Problem Formalization

The setup for the file synchronization problem is as follows. We have two files (strings) $f_{new}, f_{old} \in \Sigma^*$ over some alphabet Σ (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. We limit ourselves to a single round of messages between client and server, and measure communication cost in terms of the total number of bits exchanged between the two parties.

For a file f, we use $f[i]$ to denote the ith symbol of f, $0 \leq i < |f|$, and $f[i, j]$ to denote the block of symbols from i up to (and including) j. We assume that each symbol consists of a constant number of bits. All logarithms are with base 2, and we use $\lceil p \rceil_2$ and $\lfloor p \rfloor_2$ to denote the next larger and next smaller power of 2 of a number p.

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. Some of the most common ones are:

- The *Hamming distance* between two files f, f' of equal length is defined as the number of positions i with $f[i] \neq f'[i]$. Hamming distance is not a good model for unstructured files (as opposed to record-based data) since inserting a symbol at the beginning and deleting one at the end would result in a very large distance due to alignment issues.
- The *edit distance* (also called Levenshtein distance) is the smallest number of insertions, deletions, and changes of single symbols needed to transform one file into the other.
- 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.

There are a number of other distance measures that have been proposed; see [8], [7], [10] 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. We note that even more powerful models, such as models allowing block copies and deletions of blocks, are much harder to analyze and known upper bounds for these cases are often significantly worse [8], [7], [10].

We call a file synchronization protocol feasible if it can be implemented with a polynomial amount of computation (including the cost of decoding the current file at the receiver). A protocol is communication-efficient if it communicates at most $O(k \lg^c(n))$ bits, for some constant c, where k is the distance between the two files according to some distance measure and n is the length of the current file. (An upper bound for k may or may not be known at the start of the protocol.) We note that a lower bound of $\Omega(k \lg n))$ bits holds for all of the above distance measures. We assume that both machines have access to a random hash function, and we are interested in protocols that succeed with some fairly high probability p.

C. State of the Art

We now briefly summarize the current state of the art in file synchronization techniques; some additional discussion of related work is provided in Section IV.

There are several very strong theoretical results on the communication complexity of the file synchronization problem (sometimes also called the *document exchange* or *correlated files* problem), which establish the existence of asymptotically optimal protocols consisting of one or two rounds [24], [25], [8], [7]. Some of the results model file similarity using a very general framework based on bipartite graphs [24], while others assume various edit distance measures. However, the proposed algorithms are not implementable in practice, as they assume that the receiver can invert a hash function over a large domain in order to decode the current version of the file; this assumption appears to be fundamental to the approach. Within this framework, Orlitsky and Viswanathan [26] also showed a relationship between Error Correcting Codes for noisy channels and file synchronization that may be on some level related to our erasure-based approach.

The already mentioned *rsync* algorithm uses a single round of communication, consisting of a request by the machine holding the outdated copy, and the encoded reply from the machine holding the current copy. A more detailed description is given in Section II. As *rsync* is widely used, it 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.

A number of authors have proposed multi-round algorithms for file synchronization based on divide-and-conquer approaches. The earliest such result in [33] in fact predates *rsync*, and subsequently a number of such algorithms have been proposed and analyzed [8], [7], [10], [25], [15], [37], [22]. The algorithms can be efficiently implemented (i.e., do not require inverting a hash function), and most can be shown to be communication-efficient with respect to one of the common file distance measures. A simple example of such an algorithm and its analysis is given in Section III. A recent study of an optimized implementation of multi-round synchronization in [37] shows that such approaches can achieve significant improvements in bandwidth use over *rsync*, often by a factor of 2 to 3. However, none of these algorithms appear to be currently implemented in any widely used tools.

Very recent and independent work by Chauhan and Trachtenberg [6] has proposed an approach for file synchronization based on a reduction to the set reconciliation problem. The algorithm works in two rounds and achieves provable bounds with respect to certain graph-based measures, but it is not communication-efficient according to the above definition in the worst case.

Thus, the *rsync* algorithm appears to be the best single-round algorithm currently known, but there is significant room for improvements in its bandwidth use. 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. However, single-round protocols are preferable in many scenarios involving small files or large latencies, for example the web access application where a single HTML page is retrieved over a high-latency modem connection. 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 [22]. Finally, multi-round protocols tend to have higher overhead at the endpoints as they may take multiple passes over the input.

D. *Contributions of this Paper*

In this paper, we study single-round protocols for file synchronization. Our goal is to achieve significant improvements in practice over *rsync*, which is currently still the best single-round protocol. Our contributions are:

(1) We explore several techniques for optimizing and tuning the *rsync* approach, in particular use of delta compression, tuning of the hash value bit strength, use of content-dependent block boundaries, and multiple alignments of block boundaries. Our study shows that some gains over the current *rsync* implementation are possible.
(2) 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.
(3) 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.

Throughout this paper, we focus on optimizing bandwidth consumption. Communication latency is not an issue in our case since all algorithms operate in a single round. However, there are two other types of overhead that may also be significant in certain cases: (a) CPU cost due to hash computation and data structure insertions and lookups, and (b) the cost of scanning the file system and retrieving files. The latter cost, which can be very significant when synchronizing large directory trees with few changes, is the same for all discussed methods. (A significant reduction in this cost might be possible through maintenance of check sum hashes for files and directories to allow efficient identification of updated files, and is outside the scope of this paper.) The CPU cost is moderate for all our methods, but a detailed comparison using optimized data structures and hash computations remains to be done.

In the next section, we describe the *rsync* algorithm and evaluate some possible optimizations. Section III proposes and evaluates our new approach to file synchronization based on erasure codes. Finally, Section IV discusses related work, and Section V contains concluding remarks.

II. Optimizing the rsync Approach

In this section, we describe the *rsync* algorithm and then discuss and evaluate a few ideas for tuning the performance of the approach. Our conclusion is that while moderate improvements are possible, more significant ones probably require a different approach.

A. The rsync *Algorithm*

The basic approach in *rsync*, as well as most other file synchronization algorithms, is to split a file into blocks and use hash functions to compute hashes or "fingerprints" of the blocks. These hashes are then sent to the other machine, where the recipient attempts to find matching blocks in its own file. One issue is the lack of alignment between matching blocks in the two files; this is addressed by comparing received hashes not just with the corresponding block in the other file, but with all substrings of the same size. For efficiency, hashes are composed from two different hash functions, a fast but unreliable one, and a very reliable one that is more expensive to compute. The steps in *rsync* are as follows:

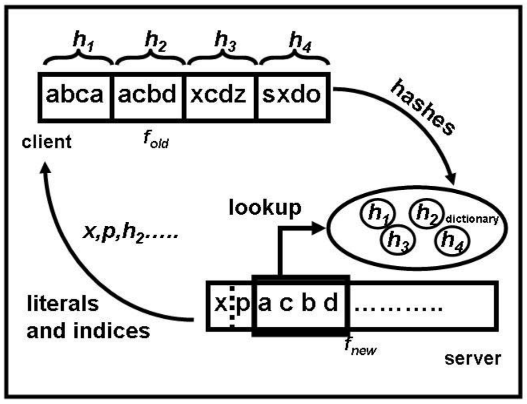

Figure II.1. The *rsync* algorithm on a small example. The client sends a set of hashes while the server replies with a stream of literals and indices identifying hashes.

1. At the client:

(a) Partition f_{old} into blocks $B_i = f_{old}[ib, (i+1)b-1]$ of some block size b.

(b) For each block B_i, compute two hashes, $u_i = h_u(B_i)$ and $r_i = h_r(B_i)$, and communicate them to the server. Here, h_u is a heuristic but fast hash function, and h_r is a reliable but expensive hash.

2. At the server:

(a) For each pair of received hashes (u_i, r_i), insert an entry (u_i, r_i, i) into a dictionary, using u_i as key.

(b) Perform a pass through f_{new}, starting at position $j = 0$, and involving the following steps:

- (i) Compute the unreliable hash $h_u(f_{new}[j, j+b-1])$ on the block starting at j.
- (ii) Check the dictionary for any block with matching unreliable hash.
- (iii) If found, and if the reliable hashes match, transmit the index i of the matching block in f_{old} to the client, advance j by b positions, and continue.
- (iv) If none found, or if the reliable hash did not match, transmit symbol $f_{new}[j]$ to the client, advance j by one position, and continue.

3. At the client:

(a) Use the incoming stream of symbols and indices of hashes in f_{old} to reconstruct f_{new}.

The process is illustrated in Figure II.1. All symbols and indices sent from server to client in steps (iii) and (iv) are also compressed using an algorithm similar to *gzip*. A checksum on the entire file is used to detect the (fairly unlikely) failure of both checksums, in which case the algorithm could be repeated with different hashes, or we simply transfer the entire file in compressed form. The reliable checksum is implemented using MD4 (128 bits), but only two bytes of the MD4 hash are used since this provides sufficient power for most file sizes. The unreliable checksum is implemented as a 32-bit "rolling checksum" that allows efficient sliding of the block boundaries by one character, i.e., the checksum for $f[j+1, j+b]$ can be computed in constant time from $f[j, j+b-1]$. Thus, 6 bytes per block are transmitted from client to server.

B. Discussion of rsync *Performance*

Clearly, the choice of block size is critical to the performance of the algorithm, but the best choice depends on the degree of similarity between the two files. Moreover, the location of changes in the file is also important. If a single character is changed in each block of f_{old}, then no match will be found by the server and *rsync* will be completely ineffective; on the other hand, if all changes are clustered in a few areas of the file, *rsync* will do well even with a large block size. Given these observations, some basic performance bounds based on block size and number and size of file modifications can be shown. However, *rsync* does not have any good performance bounds with respect to common file distance measures.

In practice, *rsync* uses a default block size of 700 bytes except for very large files where a block size of $\sqrt{n}$ is used. Decreasing the block size to 100 bytes or less is usually not practical: if one out of three hashes finds a match, this means that 18 bytes of hashes are transmitted for each discovered match, while simply applying *gzip* to the unmatched blocks might result in a reduction to about 25 bytes on average. We note that it is not difficult to find settings for the block size that perform significantly better than the *rsync* default size on particular data sets, but this by itself cannot be claimed as an improvement over *rsync* unless we get gains over a significant range.

Another way to improve performance without adding extra round-trips is to use fewer bits per hash. However, a version of the birthday paradox provides a limit to this approach: Given two files of length n, where n/b hash values are compared to the hashes of all $n-b+1$ blocks of size b in the other file, we need about $\lg(n) + \lg(n/b)$ bits per hash in order to have an even chance of not having any false match, while approximately $\lg(n) + \lg(n/b) + d$ bits are needed to get a probability less than $1/2^d$ of having any false match between the files. (We state approximate bounds here for simplicity.)

C. Some Basic Optimizations and their Performance

We now explore a few possible optimizations of the *rsync* approach. We start with two fairly obvious ones: (1) use of a better compressor for literals, and (2) a better choice of the number of bits per hash.

In the first optimization, we replace the *gzip* algorithm used for the transmission of the unmatched literals and the match tokens in *rsync* with an optimized delta compressor. A delta compressor is a tool that compresses one file called *target file* with respect to another, usually similar, file called *reference file*. The resulting *delta* is essentially a description of the differences between target and reference file, with the property that the target file can be reconstructed from the delta and the reference file. In our modification of *rsync*, the server 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 to the client. In addition, 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.

Use of a delta compressor has two advantages: First, it exploits redundancies between unmatched and matched parts of the current file; we note that the idea of exploiting this redundancy was already discussed by Tridgell [39]. Second, an optimized delta compressor may provide a more efficient way to encode offsets and indices than the tokens in *rsync*. (This also simplifies implementation and evaluation of our various methods by allowing us to sidestep the issue of how to optimize the representation and compression of these tokens.) We used the *zdelta* delta compressor [38], available at `http://cis.poly.edu/zdelta/`, which is highly efficient and achieves particularly good compression for small to medium size files. For large files beyond a few megabytes, a compressor such as *vcdiff* [13], which can capture global reorderings of substrings, would be preferable.

The second optimization chooses the number of bits in the hashes as a function of the file size (for the moment, assume both files are of similar size). In particular, we assume some upper bound on the probability of a collision, say $1/2^d$ for some d, and then use $\lg(n) + \lg(n/b) + d$ bits per hash. Of those bits, up to 32 are chosen from the weak but fast hash, and the rest from the slow hash.

We now compare basic *rsync* with a version using *zdelta* and with a version using both *zdelta* and shorter hash values for $d = 10$. For the experiments, we used the *gcc* and *emacs* data sets also used in [11], [37], consisting of versions 2.7.0 and 2.7.1 of *gcc* and 19.28 and 19.29 of *emacs*. The newer versions of *gcc* and *emacs* consist of 1002 and 1286 files, and each collection has a size of around 27 MB. In each case we measured the cost of updating all files in the older version to the newer one. Total communication cost is divided into two parts: the *read size* is the data sent from client to server (mostly hashes), while the *write size* is the data sent from server to client (mostly the delta).

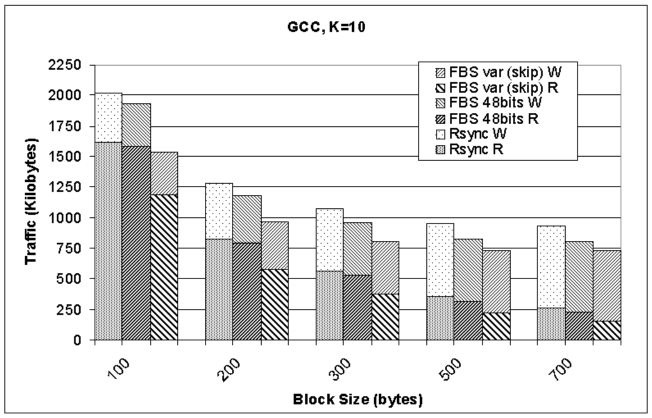

Figure II.2. Results of the first two optimizations on the *gcc* collection, for various block sizes.

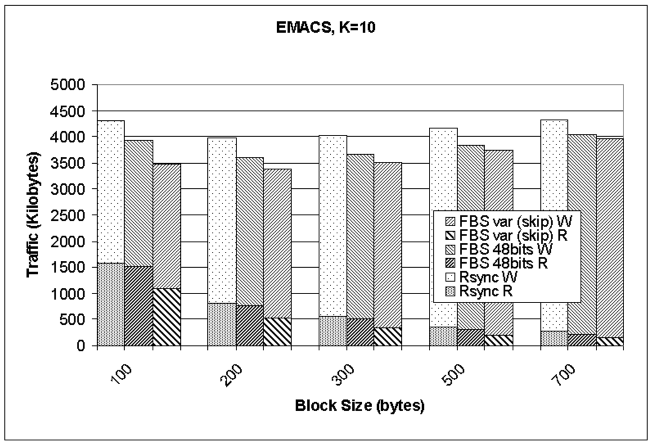

Figure II.3. Results of the first two optimizations on the *emacs* collection, for various block sizes.

The results are shown in Figures II.2 and II.3. We note that *gcc* has a much larger degree of similarity between the different versions than *emacs*. As a result, there are fairly few unmatched literals in *gcc* even with fairly large block sizes, and it is not profitable to spend extra bits on sending hashes for a smaller block size. The best block size for *gcc* is close to the default size of 700 bytes in *rsync*. For *emacs*, the best block size is fairly small, between 100 and 200, as this results in many additional matches that are not caught with larger block sizes. In general, the results show that there is no one optimal block size, and that the choice depends on the data. However, we see consistent improvements due to the two optimizations. We see improvements of 10% to 15% across the various block sizes, with improvements due to shorter hashes more prominent for small block sizes, since in this case the cost of the hashes is relatively higher.

Next, we look at the actual rate of collisions that we encounter with the shorter hash values, and their impact on performance. As in *rsync*, we assume that a 16-byte hash of the entire current file is transmitted to the client in order to detect any corruption due to false matches; in case of corruption the entire file is retransmitted encoded by *gzip*. We look at

d	% match	file coll	coll size	gzip(coll)	total
5	92.15	3.00	3.79	283909	997797
6	92.15	1.40	1.66	126869	846825
7	92.15	0.40	0.17	14845	740626
8	92.15	0.30	0.11	8998	740536
9	92.15	0.10	0.01	1627	738821
10	92.15	0.00	0.00	0	742971
5	92.14	0.20	0.24	19274	734323
6	92.13	0.00	0.00	0	720828
7	92.13	0.00	0.00	0	726476
8	92.13	0.00	0.00	0	732233
9	92.13	0.00	0.00	0	737889
10	92.13	0.00	0.00	0	743666

Table II.1

TOTAL PERCENTAGE OF FILE SIZE COVERED BY MATCHES, PERCENTAGE OF CORRUPTED FILES DUE TO HASH COLLISIONS, SIZE OF CORRUPTED FILES AS PERCENTAGE OF COLLECTION, COST OF RETRANSMISSIONS USING *gzip*, AND TOTAL COST OF THE ALGORITHM IN BYTES, FOR *gcc* WITH VARIOUS CHOICES OF d. THE FIRST 6 ROWS ARE FOR "NO-SKIP" AND THE OTHERS FOR "SKIP". THE BLOCK SIZE IS 600 BYTES.

two different implementations of the match discovery process at the server: "skip" is the implementation currently used in *rsync* where after each matched block we move our window to the end of the block, thus disallowing overlapping matches in the current file, while "no-skip" also looks for overlapping matches.

The results are shown in Table II.1. We see that as expected "skip" has significantly fewer collisions and file corruptions than "no-skip" and also fewer than predicted by our choice of k since it does not compare to all blocks in the current file. Thus, "skip" is the better option also in terms of bandwidth as it allows use of shorter hashes.

D. Variable-Size Blocks

Our next idea for improving performance is to use variable-size instead of fixed-size blocks. In particular, we evaluate the use of Karp-Rabin fingerprints [12] to determine the block boundaries, inspired by recent work [30], [34], [21], [9] that uses these techniques in other scenarios. This is done by moving a small window (e.g., of size 20 bytes) over each file. For each byte position of the window, we hash the content using a simple random hash function (not identical to the block hash). If the hash value is $0 \bmod b$ (say, $b = 256$), then we introduce a block boundary at the end of the current window.

We use this technique to partition both f_{old} and f_{new}. The purpose of the small window is to define block boundaries in a content-dependent manner. Thus, when a substring in one file contains a block boundary, then if the same substring also appears in another file, it will also contain the same block boundary. The advantage of this technique for file synchronization is that we do not have to compare each hash from f_{old} to all alignments in f_{new}, but only to those corresponding to blocks in f_{new}. Thus, the number of bits per hash can be reduced by $\lg(b)$ to a total of $2\lg(n/b) + d$ for expected block size b. Since b is typically a few hundred bytes,

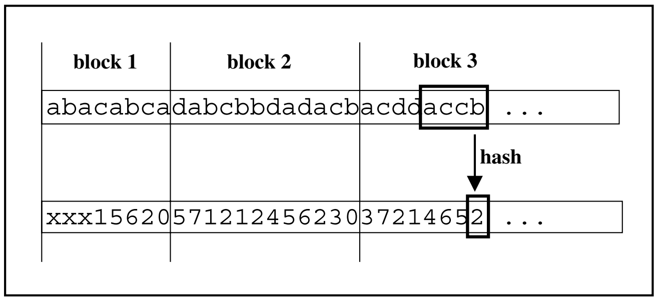
block 1
block 2
block 3
abacabcadabcbbdadacbacddaccb ...
hash
xxx15620571212456230372146 52 ...

Figure II.4. Use of Karp-Rabin fingerprints to partition a file into blocks. In this case, a window of size four bytes is moved over the file and at each position a hash $h()$ of the window is computed. Hash values are in the range $\{0, \ldots, 7\}$, and a block ends whenever we have a hash value of $0 \bmod 8$. Thus, the expected block size is 8 bytes unless there are repetitive patterns in the file.

this can result in nontrivial reductions in the cost of sending the hashes.

We experimented with two implementations. In one, we defined block boundaries as above, by introducing a block boundary at the end of the current window if the window hashes to $0 \bmod b$. In the second implementation, we use overlapping blocks by including both the boundary window to the right and to the left in each block. We also experimented with an alternative partitioning rule proposed in [32] that guarantees a lower variation in block sizes, but this did not result in any improvements.

E. Matches with Half-Block Alignment

We studied one other optimization that goes a little beyond the standard *rsync* framework. The goal is to try to address two common shortcomings in *rsync*: (1) Suppose we have hashes for two consecutive blocks that do not find a match at the server, but if we had a hash for a block of the same size that goes from the middle of the first block to the middle of the second block, it might be possible to find a match. In general, we would like to be able to find matches of large enough size that go across the block boundaries, at least for a selected set of alignments. (2) Having identified a match of one block, we should be able to efficiently extend such a match, say, into one half of the neighboring block, even if the whole neighboring block does not find a match. This is basically the idea behind the *continuation hashes* proposed in [37], that far fewer bits are needed if a hash is only compared to one block position in the other file, in this case the position adjacent to a known match.

However, implementing these ideas in a single round is tricky, and we can not get everything we want. We take the following approach: We partition the client file into blocks of fairly small size b' (say, half the size b that we would usually select under *rsync*), but send far fewer bits per hash (only about half as many). At the server, we look for matches in the other file, but we only accept matches that are part of a sequence of at least 2 consecutive matches. The reason is that the number of hash bits per single block is not large enough to identify isolated matches with confidence. To get a probability of less than $1/2^d$ of a false match in the file, we need to satisfy two conditions on the number of hash bits h per block:

$$2h \geq \lg(n) + \lg(n/b) + d \quad \text{and} \quad h \geq \lg(n/b) + d.$$

The first condition assures that two consecutive block matches suffice to identify a valid match, while the second condition assures that we can extend a match by single blocks to the left and right without too much danger of a false match. We note that instead of choosing blocks of half the usual size, other settings are possible to recognize various other alignments not exploited by *rsync*, but we did not find significant benefits in this. The above algorithm, which we refer to as *half-block alignment*, is basically a fairly crude way to exploit the fact that matches in files are clustered and that adjacent blocks in one file are more likely to match with adjacent blocks in the other file than with blocks that are far away from each other. We implemented this method for $d = 10$ and also checked that the frequency of false matches is as expected.

F. Performance of the Two Optimizations

In Figure II.5 and II.6, we compare the performance of *half-block alignment* and of the approach using variable size blocks against *rsync* and the optimized version from the previous subsection, on a range of block sizes. For the methods with variable block size, we show the expected block size and for *half-block alignment* we show twice the size b' of the small blocks in the figure (since the small blocks are half the "normal" size b to capture half-block alignments).

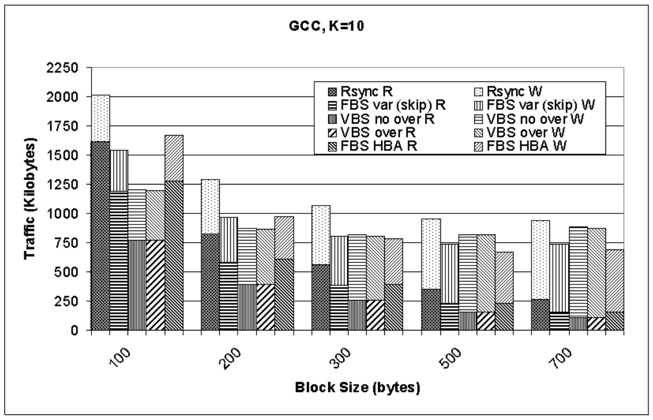

Figure II.5. Results for basic *rsync*, the optimized *rsync* from the previous subsection, the variable block size approach without and with overlap, and the *half-block alignment* protocol on the *gcc* collection, for various block sizes *b*.

We observe that the variable block-size methods do very well for small blocks. The reason is that these methods use shorter hashes, and thus benefit when the size of the hashes is significant compared to the total cost. On the other hand, methods with variable blocks tend to result in more unmatched literals, mainly due to variations in the block sizes, that dominate the savings in hash bits for larger block sizes. As suggested in [21], [30], we enforce certain minimum and maximum block sizes to deal with regular patterns in the data,

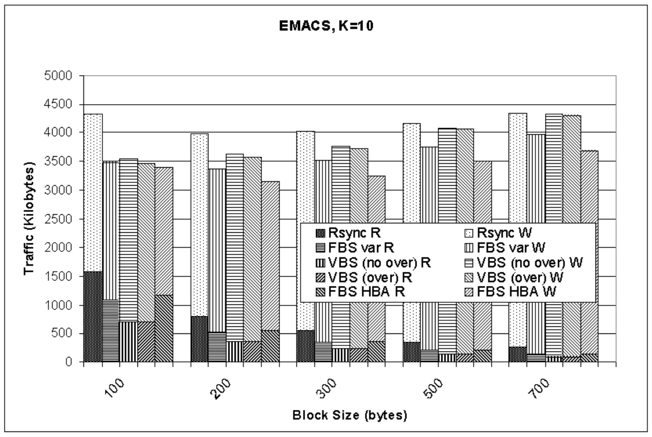

Figure II.6. Results for basic *rsync*, the optimized *rsync* from the previous subsection, the variable block size approach without and with overlap, and the *half-block alignment* protocol on the *emacs* collection, for various block sizes *b*.

but even with optimum choice of these parameters the block sizes are distributed over a certain range, and large blocks are more likely to not find a match. Moreover, we note that the savings in bits per hash for variable-size blocks are only compared to the "no-skip" version of the fixed-size method, and as seen in Table II.1 we could actually use fewer bits per hash when using the "skip" method. We observe that there is at most a very slight benefit in using overlapping instead of non-overlapping blocks.

The *half-block alignment* approach outperforms the other methods on most block sizes, with the notable exception of *gcc* for block size 100 where the second condition on the number of hash bits stated above results in a fairly high cost for the hashes. Note that this block size is not a good choice for *gcc* under any method, and is far away from the default size of 700 for *rsync* or any suitable default size. On *emacs*, we see an improvement of 5% to 10% over the next best method, and overall we see an improvement of 15% to 25% over the basic *rsync* method across the different blocks sizes, including the default size. Similar results were also obtained on several other data sets. Thus, some moderate improvements are possible through careful tuning of the *rsync* approach.

III. An Approach Based on Erasure Codes

In this section, we provide the main result of this paper, a new approach to single-round file synchronization based on the use of erasure code. Using this approach, 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 III-B contains the theoretical result obtained by converting the multi-round protocol, and Subsection III-C describes and evaluates the practical protocol.

A. 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 [33], [8], [25], [37]. (The multi-round protocols in [15], [22] are also similar, but send hashes from client to server.)

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 simply sends all unmatched blocks as literals. Figure III.1 illustrates the protocol on a small example.

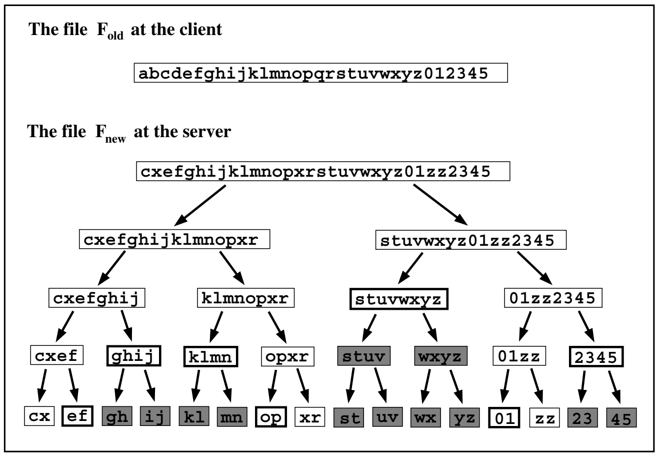

Figure III.1. The basic multi-round protocol on a small example file pair. Shown in bold outlines are blocks in the server file that find a match at the client, while blocks that have a matched ancestors are shaded. Hashes for the latter blocks are not communicated in the *basic multi-round protocol*, but would be communicated in the *complete multi-round protocol*.

Suppose we choose $b_{max} = \lfloor n/k \rfloor_2$, $b_{min} = \lg(n)$, and use hashes of size, say, $4 \lg n$ bits. 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 next 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 disjoint 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 increases the number of pieces of the old file by at most one, while each move operation may require up to three cuts and thus increases the number of pieces by at most three. Any substring s_i that is completely within one of the at most k pieces clearly also occurs in f_{old}, giving the result.

There are several possible practical optimizations to this algorithm [37], but this is not our concern. In the following, we show how to convert this algorithm into a single-round protocol with the same complexity.

B. An Efficient Single-Round Protocol

First, we define the *complete multi-round protocol* as the variation of the basic multi-round protocol from the previous subsection 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 [31] 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 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 source 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 protocol that would not be sent in the basic multi-round protocol

(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 from all hashes on a level, 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 the $2k$ 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 children of blocks that were already matched on a higher level in order to compute their hashes, and 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 with block moves 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 [8]. 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 many clients holding different outdated versions.

C. 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 that make it inferior to *rsync* and other existing protocols in practice.

- The protocol requires us to estimate an upper bound on the file distance k. This adds complexity to the implementation, and while there are efficient protocols for this, we need to make sure that we do not underestimate, since otherwise the client is unable to recover the current file. Thus, to be sure we may have to send more than needed.
- More importantly, the algorithm does not support compression of unmatched literals but essentially sends them in raw form as hashes. The performance of *rsync* and other protocols such as [37] is significantly improved through the use of compression for literals. There are some tricks that one can use to integrate compression into the algorithm, but this seems to lead either to variable-size data items in the erasure coding at the leaf level, or to severely reduced compression if we force all items to be of the same size.

To address these problems we 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 fact, similar to the way the first algorithm was obtained from the basic and complete multi-round algorithms by adding erasure hashes, the new algorithm can be obtained by adding erasure hashes to a multi-round algorithm similar to those in [15], [22] that sends hashes from client to server. (These algorithms are essentially multi-round versions of *rsync*, which explains the similarity of our algorithm below to *rsync*.) In the following description, note that the first three steps are identical to the previous algorithm but with the roles of client and server exchanged.

(1) The client partitions f_{old} recursively into blocks from size b_{max} down to b_{min}, and for each level computes all block hashes.
(2) The client applies a systematic erasure code to each level i of hashes except the top level, and computes m_i erasure hashes for each level, for some appropriate m_i discussed later.
(3) In one message, the client sends all hashes at the highest level to the server, plus the m_i erasure hashes for each level i.
(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* or in our variations of *rsync*. Thus, common blocks are identified and all unmatched literals are sent in compressed form to the client.

If we set the parameters as in the theoretical algorithm, 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 making the reasonable assumption that compression does not significantly 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. For example, if we have 50 blocks on the second highest level ($i = 2$), we might choose $r_2 = 0.2$ which means that we use $m_2 = r_2 \cdot 50 = 10$ erasure hashes on this level. Then we will be able to decode this second level successfully provided that at most 20% of the hashes on the highest level did not find a match in f_{old}. Thus, 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 experimented with a number of choices of b_{max}, b_{min}, and the r_i.

We implemented the algorithm based on 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 in most cases. We chose the number of bits per hash as $\lg(n) + \lg(y) + k$ for $k = 10$, where y is the total number of hashes (top-level and erasure hashes) sent from client to server.

We integrated two additional optimizations into the implementation, both of which are completely orthogonal to the erasure coding approach and very simple to add. First, we integrated *decomposable hashes*, first proposed in [33] and recently rediscovered in [37], which allow 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. In particular, we apply erasure coding only to left siblings and compute the hashes of the right siblings at the server after decoding left siblings. Second, we added the half-block alignment approach from Subsection II-E. Alternatively, we could also integrate *continuation hashes* [37] instead of half-block alignments to get additional minor improvements, but this has not been implemented yet.

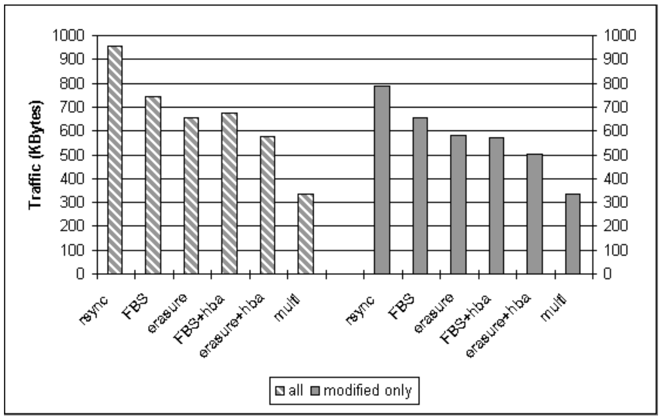

Figure III.2. Results on *gcc* data with the best possible parameter settings for the following methods (from left to right): *rsync*, the optimized *rsync* from Subsection II-C, the erasure-code approach with decomposable hashes, the half-block alignment approach, the erasure-code approach with decomposable hashes and half-block alignment, and the optimized multi-round approach from [37]. Results on the left are for all files, while results on the right are only for changed files in the collection.

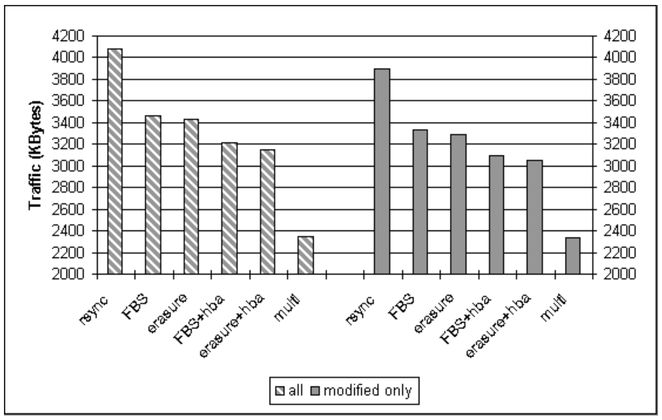

Figure III.3. Results on *emacs* data with the best possible parameter settings for the following methods (from left to right): *rsync*, the optimized *rsync* from Subsection II-C, the erasure-code approach with decomposable hashes, the half-block alignment approach, the erasure-code approach with decomposable hashes and half-block alignment, and the optimized multi-round approach from [37]. Results on the left are for all files, while results on the right are only for changed files in the collection. Note that the y-axis starts from 2000 KB.

In Figures III.2 and III.3 we show experimental results comparing the erasure-code approach to the various optimized methods discussed in this paper. For each method, we show the best result that we obtained. The result from [37] essentially provides a upper bound on what we can realistically hope to gain with a single-round approach, barring further breakthroughs in techniques. We see that the erasure-code based approach does about 10% better than the other approaches on *gcc*, but provides only negligible improvements on *emacs*. Recall that the two versions of *gcc* are more similar to each other than the *emacs* versions. The best block size for *gcc* was fairly large, around 700% bytes, since most matches were already found at such large block sizes and the cost of using smaller blocks would outweigh any benefits due to additional matches. The erasure-code approach allows us to effectively use smaller blocks in *gcc* without paying the full cost of each block, since many matches already occur at higher levels.

We also show results for both collections were we only consider files that differ in the two versions. The motivation for this is that even in scenarios that use a single-round protocol, there is often an extra initial round of communication where client and server exchange lists of file names, last modification dates, and possibly file checksums in order to identify the files that need to be synchronized. As we see the results are not substantially different for this case. We note that one problem with the erasure-based approach is how to select the optimum setting of the parameters, i.e., the block sizes and the thresholds r_i. We used for each method the best setting that we could find for each collection; this maybe gives erasure coding an advantage since there are more parameters and thus more knobs that we can tweak to optimize performance. It is an open problem to come up with good rules for choosing parameters, and possibly one could use an extra initial round to exchange statistics that allow a good choice of parameters for each file.

The erasure-based approach presented here allows for a number of additional optimizations and could be used to design other new algorithms. For example, one could design protocols with two or more round-trips that first send a very small number of erasure hashes, and then send additional erasure hashes in the next round if needed to successfully decode the lower levels. Thus, instead of processing one level of blocks in each round, as the previous multi-round approaches do, this approach would simultaneously add redundancy to several levels until decoding succeeds on the other side. This idea could potentially also be generalized into a *digital fountain* approach [5] for broadcasting updated content to clients that may have different old versions.

One could also try to combine erasure coding with error correcting codes where the positions of corrupted symbols are unknown to the decoder. The current algorithm chooses the bit strength of the hashes such that it is unlikely that there are any false matches at all in the entire file. This could be relaxed to a bit strength such that most matches are correct (even though there are likely to be some false matches in a larger file), if we use an error correcting code that can correct a few corrupted hashes due to false matches at the parent level. Thus, in this scenario a number of hashes would be known to be lost (erasure case), while a few others in unknown positions would be corrupted (ECC case). We believe there are other interesting protocols that can be designed with our approach.

IV. Related Work

In the following, we give a brief summary of related work. The *rsync* algorithm proposed by Tridgell and MacKerras is described in [39], [41], and is the basis of the very widely used *rsync* open source tool. There are a number of theoretical studies of the file synchronization problem [8], [7], [23], [24]. In particular, Orlitsky [23], [24] presents almost tight bounds for the problem with varying numbers of communication phases, under some assumptions about the assumed file distance metric. As explained, these results typically require exponential time for decoding; while this is allowable under the standard model for communication complexity [14], it makes the algorithms impractical. Within this framework, [26] discusses a relationship between Error Correcting Codes and file synchronization.

Various practical multi-round algorithms are proposed in [33], [8], [7], [10], [25], [15], [37], [22]. These algorithms are based on recursive partitioning of unmatched blocks, mostly in a breadth-first manner with the exception of [10]. The algorithms in [15], [22] send hashes from client to server, while the others send hashes in the other direction. Experimental results for multi-round algorithms are provided in [15], [25], [37], [22].

Some available open source tools for delta compression are described in [13], [16], [38], and an overview of delta compression and file synchronization techniques and their applications is given in [36]. Note that delta compression can be seen as a special case of file synchronization where the outdated file is known to the encoder.

A number of authors have studied problems related to identifying disk pages, files, or data records that have been changed or added or deleted, or that differ between two or more replicas; see, e.g., [1], [4], [17], [18], [19], [20], [27], [33]. Common approaches for these problems are based on hashing or coding techniques. The problem setup differs from ours in that data is assumed to be partitioned into fixed units such as pages, records, or files, that are treated as atomic. Recent work on the *set reconciliation problem* in [20], [2], [35] also falls into this category. Very recent independent work by Chauhan and Trachtenberg [6] shows how to use set reconciliation techniques for file synchronization.

Hash-based techniques similar to *rsync* have been explored by the OS and Systems community for purposes such as compression of network traffic [34], distributed file systems [21], distributed backup [9], and web caching [30]. These techniques use string fingerprinting techniques [12] to partition a data stream into blocks, as we did in Subsection II-D.

V. Conclusions and Open Questions

In this paper, we have studied single-round protocols for file synchronization. Our main contribution has been a new approach 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 slight gains once we fully optimize the implementation and parameter settings. We hope 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.

It would be interesting to explore the trade-off between bandwidth consumption and the number of round-trips. We suspect that an approach with two or three rounds might do significantly better than the single-round approaches in this paper. Closely related to this problem is how to adaptively choose the best algorithm and parameter setting for a given pair of files, say by exchanging samples or other statistics at the start of the synchronization. In addition, there are a number of interesting open theoretical 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.

ACKNOWLEDGMENTS

This work was supported by NSF CAREER Award CCR-0093400, NSF ITR Awards CNS-0325777 and IDM-0205647, and the Wireless Internet Center for Advanced Technology (WICAT) at Polytechnic University.

REFERENCES

[1] K. Abdel-Ghaffar and A. El Abbadi. An optimal strategy for comparing file copies. *IEEE Transactions on Parallel and Distributed Systems*, 5(1):87–93, January 1994.

[2] S. Agarwal, D. Starobinski, and A. Trachtenberg. On the scalability of data synchronization protocols for PDAs and mobile devices. *IEEE Network Magazine, special issue on Scalability in Communication Networks*, July 2002.

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

[4] D. Barbara and R. Lipton. A class of randomized strategies for low-cost comparison of file copies. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):160–170, April 1991.

[5] J. Byers, M. Luby, and M. Mitzenmacher. A digital fountain approach to the reliable distribution of bulk data. *IEEE Journal on Selected Areas in Communications*, pages 1528–1540, October 2002.

[6] V. Chauhan and A. Trachtenberg. Reconciliation puzzles. In *Proc. of the IEEE GlobeCom Conference*, November 2004. to appear.

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

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

[9] L. Cox, C. Murray, and B. Noble. Pastiche: Making backup cheap and easy. In *Proc. of the 5th Symp. on Operating System Design and Implementation*, December 2002.

[10] A. Evfimievski. 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, January 1998.

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

[12] R. Karp and M. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[13] D. Korn and K.-P. Vo. Engineering a differencing and compression data format. In *Proceedings of the Usenix Annual Technical Conference*, pages 219–228, June 2002.

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

[15] J. Langford. Multiround rsync. January 2001. Unpublished manuscript.

[16] J. MacDonald. File system support for delta compression. MS Thesis, UC Berkeley, May 2000.

[17] T. Madej. An application of group testing to the file comparison problem. In *Proc. of the 9th Int. Conf. on Distributed Computing Systems*, pages 237–243, June 1989.

[18] J. Metzner. A parity structure for large remotely located replicated data files. *IEEE Transactions on Computers*, 32(8):727–730, August 1983.

[19] J. Metzner. Efficient replicated remote file comparison. *IEEE Transactions on Computers*, 40(5):651–659, May 1991.

[20] Y. Minsky, A. Trachtenberg, and R. Zippel. Set reconciliation with almost optimal communication complexity. Technical Report TR2000-1813, Cornell University, 2000.

[21] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proc. of the 18th ACM Symp. on Operating Systems Principles*, pages 174–187, October 2001.

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

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

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

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

[26] 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.

[27] C. Park and J. J. Metzner. Efficient location of discrepancies in multiple replicated large files. *IEEE Transactions on Parallel and Distributed Systems*, 13(6):597–610, June 2002.

[28] B. Pierce. Unison file synchronizer. `http://www.cis.upenn.edu/~bcpierce/unison/`.

[29] 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.

[30] S. Rhea, K. Liang, and E. Brewer. Value-based web caching. In *Proc. of the 12th Int. World Wide Web Conference*, May 2003.

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

[32] S. Schleimer, D. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proc. of the 2003 ACM SIGMOD Int. Conf. on Management of Data*, pages 76–85, 2003.

[33] 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.

[34] N. Spring and D. Wetherall. A protocol independent technique for eliminating redundant network traffic. In *Proc. of the ACM SIGCOMM Conference*, 2000.

[35] D. Starobinski, A. Trachtenberg, and S. Agarwal. Efficient PDA synchronization. *IEEE Transactions on Mobile Computing*, 2(1), 2003.

[36] T. Suel and N. Memon. Algorithms for delta compression and remote file synchronization. In Khalid Sayood, editor, *Lossless Compression Handbook*. Academic Press, 2002.

[37] 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.

[38] 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.

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

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

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