Algorithms for Low-Latency Remote File Synchronization

Hao Yan[*]
CIS Department
Polytechnic University
Brooklyn, NY 11201
hyan@cis.poly.edu

Utku Irmak[†]
Yahoo! Inc.,
2821 Mission College Blvd,
Santa Clara, CA 95054
uirmak@yahoo-inc.com

Torsten Suel
CIS Department
Polytechnic University
Brooklyn, NY 11201
suel@poly.edu

Abstract

The remote file synchronization problem is how to update an outdated version of a file located on one machine to the current version located on another machine with a minimal amount of network communication. It arises in many scenarios including web site mirroring, file system backup and replication, or web access over slow links. A widely used open-source tool called rsync *uses a single round of messages to solve this problem (plus an initial round for exchanging meta information). While research has shown that significant additional savings in bandwidth are possible by using multiple rounds, such approaches are often not desirable due to network latencies, increased protocol complexity, and higher I/O and CPU overheads at the endpoints.*

In this paper, we study single-round synchronization techniques that achieve savings in bandwidth consumption while preserving many of the advantages of the rsync *approach. In particular, we propose a new and simple algorithm for file synchronization based on* set reconciliation *techniques. We then show how to integrate sampling techniques into our approach in order to adaptively select the most suitable algorithm and parameter setting for a given data set. Experimental results on several data sets show that the resulting protocol gives significant benefits over* rsync*, particularly on data sets with high degrees of redundancy between the versions.*

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 In particular, we have two versions of each file, an outdated file f_{old} held by one machine, the *client*, and a current file f_{new} held by another machine, the *server*. Neither machine knows the exact content of the file held by the other machine. Periodically, the client might initiate a synchronization operation that updates its file f_{old} to the current version f_{new}. Or alternatively, a synchronization operation is triggered whenever f_{old} is accessed at the client. If the amount of data is large, or the network fairly slow, then it is desirable to perform this synchronization with a minimum amount of communication.

This *remote file synchronization problem* arises in a number of applications. For example, we might want to mirror busy web and ftp sites (including sites that distribute new versions of software), or synchronize personal files between two different machines, say a machine at home and one at work (or a mobile device). Also, content distribution networks, search engines, and other large web services often need to update large collections of data within their distributed architectures. In all of these applications, there are usually significant similarities between successive versions of the files, and we would like to exploit these in order to decrease the amount of data that is transmitted. If the new files differ only very slightly from their previous versions, then the cost should be very low, while for more significant changes, more data may have to be transmitted.

Remote file synchronization has been studied extensively over the last ten years, and existing approaches can be divided into single-round and multi-round protocols. Single-round protocols [32, 12] are preferable in scenarios involving small files and large network latencies (e.g., web access over slow links). The best-known single-round protocol is the algorithm used in the widely used *rsync* open-source tool for synchronization of file systems across machines [32]. (The same algorithm has also been implemented in several other tools and applications.)

However, in the case of large collections and slow networks it may be preferable to use multiple rounds to further reduce communication costs, and a number of protocols have been proposed; see [27, 6, 5, 9, 22, 16, 30, 20]. Experiments have shown that multi-round protocols can provide significant bandwidth savings over single-round protocols on typical data sets [30, 16]. However, multi-round protocols have several disadvantages in terms of protocol complexity and computing and I/O overheads at the two endpoints; this motivates the search for single-round protocols that transmit significantly less data than *rsync* while preserving its main advantages.

In this paper, we focus on the single-round case. We propose an algorithm for file synchronization based on the use of *set reconciliation* techniques [18]. The communication cost of our algorithm is often significantly smaller than that of *rsync*, especially for very similar files. However, the basic version of our algorithm assumes knowledge of some (preferably fairly tight) upper bound on the number of “differences” between the versions. To address this issue, we then study how to integrate sampling techniques into the basic protocol in order to estimate this difference, and to choose suitable approaches and parameters based on the underlying data set.

Note a few assumptions in our basic setup. We assume that collections consist of unstructured files that can 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 blocks or bit positions or that assume fixed record boundaries do not work (though such techniques are potentially useful, e.g., for identifying which files have been changed). Note also that the problem would be much easier if all updates to the files are saved in a log that can be transmitted to the other machine, or if the machine holding the current version also has a copy of the outdated one. However, in many scenarios this is not the case. We are also not concerned with issues of consistency in between

[*]Work supported by NSF Awards CNS-0325777 and CCR-0093400, and the New York State Center for Advanced Technology in Telecommunications (CATT) at Polytechnic University.

[†]This work was performed while this author was a graduate student at Polytechnic University.

synchronization steps, or with the question of how to resolve conflicts if updates can be concurrently performed at several locations (see [2, 24] for a discussion). We assume a simple two-party scenario where it can be easily determined which version is the most current one.

1.1 State of the Art and Related Work

Previous theoretical studies of the communication complexity of the file synchronization problem have shown asymptotically tight bounds using one or more rounds [21, 22, 6]. However, the proposed optimal algorithms are not implementable in practice, as they require the receiver to invert a hash function over a large domain in order to decode the new version of the file. In practice, the most widely used protocol for file synchronization is the *rsync* algorithm [32], which is employed within a widely used open-source tool of the same name. A single round of messages is exchanged for each file, where the client sends a hash value for each block of some fixed size in the old file, which the server then uses to compress the new file. The *rsync* algorithm does not achieve strong provable bounds on communication costs, but seems to perform well in practice.

Several researchers have proposed multi-round algorithms based on a divide-and-conquer approach. The algorithms first send hashes for large blocks, and then recursively divide any unmatched blocks into smaller blocks and send hashes for these, until either a block with matching hash is found on the other side or the recursion is terminated. The earliest such algorithm [27] predates *rsync*, and subsequently a number of such algorithms have been proposed [6, 9, 22, 16, 30]. The algorithms can be efficiently implemented, and their communication costs are usually within a (poly)logarithmic factor of optimal under common measures of file similarity. As shown in [30], a highly optimized implementation can give significant improvements over *rsync*.

Thus, there is a trade-off between the number of communication rounds and the bandwidth consumption. Extra rounds may increase communication latencies for small files, though these latencies can be hidden on larger collections. Moreover, multi-round protocols may require multiple passes over the data at the endpoints, and typically also have a more complex control structure. In practice, single-round protocols such as *rsync* are implemented in two rounds, an initial exchange of meta data (e.g., directory data and MD5 hashes for all files), and then the actual *rsync* algorithm. Our algorithms also take this form, maybe best described as "one to two rounds".

There are two other recent results that try to improve on *rsync* by using only one or two rounds. First, [12] studies several optimizations to the *rsync* approach, and also proposes a new approach to file synchronization based on erasure codes. In particular, it is shown how to simulate a basic multi-round protocol in a single round of messages, assuming an upper bound on the difference between the files.

The second result, very closely related to our approach, is the protocol in [1], which is also based on the use of *set reconciliation* techniques. The *set reconciliation* problem was originally introduced in [18] in the context of synchronizing structured data items (e.g., database records). The algorithm in [1] is a two-round protocol, and it achieves provable bounds with respect to certain measures, but it also has several shortcomings that we think make it less practical in many scenarios. We discuss the protocol in [1] and its relationship to our work in more detail later, after providing the necessary background.

There is a significant amount of related work by the OS and Systems communities that attempts to detect redundancies in order to reduce storage or transmission costs, such as the *Low Bandwidth File System* [19], storage and backup systems [23, 7, 15, 31, 8], value-based web caching [25, 13], and compression of network traffic [28]. Most of these schemes operate on a lower level within network or storage protocols and thus have no notion of file or document versions. The approaches usually rely on content-based partitioning techniques such as [14, 26, 31], rather than the fixed-size blocks in *rsync*, in order to divide the data into blocks.

Finally, a number of researchers have used sampling to estimate similarities between files [4, 33, 10, 31]. We employ fairly standard sampling techniques, and focus on how to engineer these techniques to fit into our application.

2 Contributions of this Paper

We study single-round algorithms for the remote file synchronization problem, and propose new protocols that result in significant bandwidth savings over previous approaches. In particular, our contributions are:

(1) We present a new single-round algorithm for file synchronization based on set reconciliation.

(2) We explore the use of random sampling techniques in order to engineer a practical protocol based on the new algorithm. In particular, we use sampling in order to estimate the cost of the protocol, and to choose the best combination of techniques and parameter settings for the current data set. We also discuss how these techniques can be implemented in a setup very similar to the current *rsync* system architecture.

(3) We evaluate our protocols on several data sets. The results show that our protocols achieve significant improvements over previous single-round protocols for data sets with significant similarity between versions.

The remainder of this paper is organized as follows. We first provide some technical background on *rsync*, set reconciliation, and content-dependent string partitioning. In Section 4 we describe our basic algorithm and evaluate its performance under certain idealized assumptions. In Section 5, we employ sampling techniques to derive a realistic protocol that adapts to characteristics of different data sets. Finally, Section 6 provides some concluding remarks.

3 Technical Preliminaries

3.1 The *rsync* Algorithm

The basic idea in *rsync* and most other synchronization algorithms is to split a file into blocks and use hash functions to compute hashes or "fingerprints" of the blocks. These hashes are sent to the other machine, which looks for matching blocks in its own file. In *rsync*, the client splits its file into disjoint blocks of some fixed size b and sends their hashes to the server. Note that due to possible misalignments between the files, it is necessary for the server to consider every window of size b in f_{new} for a possible match with a block in f_{old}. The complete algorithm is as follows (slightly simplified):

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 a hash value $h_i = h(B_i)$ and communicate it to the server.

2. At the server:

(a) For each received hash h_i, insert an entry (h_i, i) into a dictionary, using h_i as key.

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

   (i) Compute the hash $h(f_{new}[j, j+b-1])$ on the block starting at j, and check the dictionary for any matching hash.

   (ii) If found, transmit the index i of the matching block in f_{old} to the client, increase j by b, and continue.

   (iii) If none found, transmit the symbol $f_{new}[j]$ to the client, increase j by one, and continue.

There are various additional optimizations in the algorithm. All symbols and indices sent to the client in steps (ii) and (iii) are also compressed using an algorithm similar to *gzip*; this is crucial for performance in practice. A 128-bit hash on the entire file is used to detect any (fairly unlikely) collisions in the block hashes, in which case we resend f_{new} in compressed form. The block hash function $h()$ itself is carefully chosen for efficiency and robustness. In particular, a "rolling" hash function is used such that the hash of $f[j+1, j+b]$ can be computed in constant time from that of $f[j, j+b-1]$.

A critical issue in *rsync* as well as other single-round algorithms is the choice of the block size b, and the best choice depends on both the number and granularity of changes between the versions. 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 changes are clustered in a few areas of the file, *rsync* will do well even with a large block size. Fortunately, the latter case is common in many applications. In practice, *rsync* uses a default block size of 700 bytes (except for large files where the square root of the file size is used), though this is clearly not the best choice for every data set.

Each block hash sent to the server in *rsync* has a size of 48 bits. Some improvements can be obtained by choosing fewer bits per hash, depending on the sizes of the files; see, e.g., [32, 30, 12]. Given two files of length n, where each hash in f_{old} is compared to the hashes of all $n - b + 1$ blocks of size b in f_{new}, we need about $\lg(n) + \lg(n/b)$ bits per hash to have an even chance of not having any false match, while about d more bits are needed to get a probability less than $1/2^d$ of having any false match between the files.

Finally, we discuss the structure of the *rsync* open source tool, based on [34]. Strictly speaking, *rsync* involves two roundtrips. First, the client requests synchronization of a directory, and the server replies with some meta information which allows the client to decide which files need to be updated. Next, synchronization is performed as described. This is implemented in a highly pipelined fashion, where the two endpoints are implemented as separate independent processes that consume and produce streams of data. Thus, round-trips are not incurred on a per-file basis, and it might seem reasonable to add some extra rounds to obtain additional bandwidth savings. However, this would require either a larger fixed set of consumer/producer processes, or we would have to keep some state for each file. In our approach, we adhere to this simple structure, with a first round for exchanging meta data, and one subsequent round where the actual synchronization is performed.

3.2 The Set Reconciliation Problem

We now discuss the *set reconciliation problem* defined in [18, 29] and its relation to file synchronization. In the *set reconciliation problem*, we have two machines A and B that each hold a set of integer values S_A and S_B, respectively. Each host needs to find out which integer values are in the intersection and which are in the union of the two sets, using a minimum amount of communication. The goal is to use an amount of communication that is proportional to the size of the *symmetric difference* between the two sets, i.e., the number of elements in $(S_A - S_B) \cup (S_B - S_A)$. A protocol based on characteristic polynomials [18] achieves this with a single message. We define the characteristic polynomial $\chi_S(Z)$ of a set $S = \{x_1, x_2, ..., x_n\}$ as the univariate polynomial

$$\chi_S(Z) = (Z - x_1)(Z - x_2)...(Z - x_n). \qquad (1)$$

An important property of the characteristic polynomial is that it allows us to cancel out all terms corresponding to elements in $S_A \cap S_B$, by considering the ratio between the characteristic polynomials of S_A and S_B

$$\frac{\chi_{S_A}(Z)}{\chi_{S_B}(Z)} = \frac{\chi_{S_A \cap S_B}(Z) \cdot \chi_{\Delta_A}}{\chi_{S_A \cap S_B}(Z) \cdot \chi_{\Delta_B}} = \frac{\chi_{\Delta_A}(Z)}{\chi_{\Delta_B}(Z)}. \qquad (2)$$

In order to determine the set of integers held by the other party, it suffices to determine the ratio of the two polynomials. As observed in [18], if we know the results of evaluating both polynomials at only k *evaluation points*, where k is the size of the symmetric difference, then we can determine the coefficients of $\chi_{\Delta_A}(Z)/\chi_{\Delta_B}(Z)$). By factoring $\chi_{\Delta_A}(Z)$ and $\chi_{\Delta_B}(Z)$ we can recover the elements of Δ_A and Δ_B. Thus, if the difference between the two sets is small, then only a small amount of data has to be communicated.

This problem was initially studied in the context of synchronizing structured data items: If we represent each data record by an integer hash, then we can determine which records already exist at the recipient. If each hash value consists of x bits, then all arithmetic can be performed modulo 2^x, and thus each evaluation point can be communicated in x bits. If a record does not exist at the recipient, then the entire record is transmitted in the next step. However, in file synchronization, we want to exploit similarities between different versions even if the file has changed to some degree.

Recent work in [1] proposed a new algorithm for file synchronization, called *reconciliation puzzles*, that uses set reconciliation as a main ingredient. Each machine converts its file (string) into a multi-set of overlapping pieces, where each piece is created at every offset of the file according to a predetermined mask. The hosts also create a modified de Bruijn digraph to enable decoding of the original string from the multi-set of pieces: The correct Eulerian path on this digraph determines the ordering of the pieces in the original string.

To reconstruct f_{new}, the client needs to know all the pieces of f_{new}, plus the Eulerian path on the digraph. Both sides transform their sets of pieces into integer values by concatenating each piece with the number of times it occurs and hashing the resulting string. Then the set reconciliation protocol from [18] is used to reconcile the sets of hash values, at which point the server knows which pieces are unknown to the client and thus need to be transmitted. If the two versions are very similar, then they will have most of their hash values in common, and the set reconciliation algorithm only needs to transmit a very small amount of data. Finally, a compact encoding of the Eulerian path is transmitted to the client together with the missing pieces, resulting in a two-round protocol.

3.3 Content-Dependent File Partitioning

Using a fixed block size as in *rsync* means that the recipient has to compute hashes for all possible alignments in its file in order to find a match. This requires not only more computation, but also more bits in each hash. In the set reconciliation approach, the files cannot be simply partitioned into fixed size blocks for the reconciliation step; chances are that the resulting sets would be almost disjoint due to different alignments of the common content in both versions. In [1], this problem is avoided by creating many more blocks (pieces) from each file in order to capture every possible alignment. However, this significantly increases the size of the resulting sets and graphs, resulting in extra cost.

We would like a partitioning that allow us to find many common blocks in similar files and that can be locally applied to each file. One such technique, based on Karp-Rabin fingerprints [14], has been extensively used to identify redundant data during storage or transmission [19, 7, 15, 31, 25, 13]. The basic idea is very simple: We hash all substrings of some fixed length c, say $c = 20$, to integer values, and define a block boundary before position i of the file f if the substring $f[i, i + c - 1]$ hashes to a value $s \bmod w$, say for $w = 200$ and some s. Then we use the blocks defined by these boundaries, which have an average length of about w (under certain assumptions). Since boundaries are chosen in a local manner, any large substrings common to both files are partitioned in the same way and result in identical blocks. After identifying the boundaries using the above method, we then hash the resulting blocks to integers as part of the synchronization protocols.

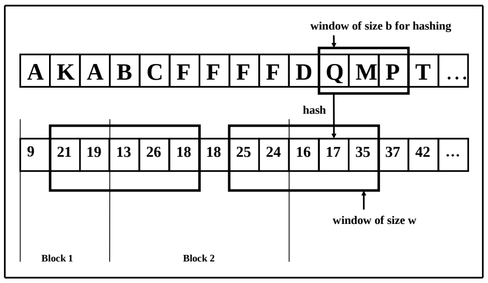

Figure 3.1. Example of the file partitioning technique in [31]. We are hashing windows of size $c = 3$, and then selecting block boundaries based on these hashes for $w = 2$.

More recently, several new partitioning techniques have been proposed, including the *winnowing* technique in [26], and the approach in [31] which we refer to here as *2-way min*. We focus on the latter technique, defined as follows: We again hash all substrings of some fixed length c to integer values. We now define a block boundary before position i of the file f if the hash $h(f[i, i + c - 1])$ associated with position i is strictly smaller than the hash values of the w preceding and following positions. In other words, we choose those positions where the hash assumes a local minimum (or maximum). The method results in an average block size of $2w$ and is illustrated in Figure 3.1.

Most applications, such as [19, 25, 7], use the older method to select the boundaries. However, analysis in [3] and unpublished experiments indicate that the methods in [26, 31] outperform this method in redundancy detection.[1] To verify this expectation, we compared the three methods in terms of their tradeoff between average block size and the amount of redundancy that is detected between the two versions. The results are shown in Figure 3.2.

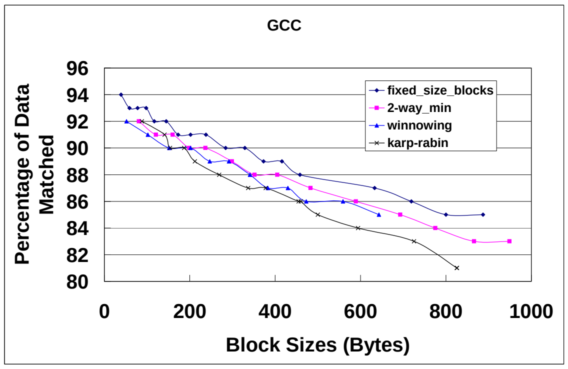

Figure 3.2. Comparison of different partitioning techniques on various block sizes on the *gcc* data set.

Clearly, as block size decreases all methods can identify more redundancy, but a small block size requires more hashes to be sent from server to client. Note that 2-*way min* does better than *winnowing*, which does better than the basic approach, confirming the analysis in [3] on real data. Overall, fixed block size does best, as should be expected, but note that we need more bits per hash value in this case, which eliminates the advantage. We also experimented with other data sets, and with several other methods not described here that performed almost as good as *2-way min* (but none did better).

Thus, we focus on the 2-*way min* method in our experiments, but any of the others would also work at slightly decreased performance. Note that these methods can be implemented at speeds of tens to hundreds of MB per second through careful optimization of the hash computation [31].

We also experimented with two different types of blocks once the boundaries are identified: Non-overlapping blocks, and overlapping blocks where each block is extended c characters into the next block (putting the window with the locally minimal hash into the intersection). Blocks with significant overlap are more suitable for use with set reconciliation techniques, since each block can only be followed by another block that "fits", i.e., that starts with the last c characters of the previous block. In terms of the algorithm in [1], this implies that the resulting digraph is very sparse and has few Eulerian paths, allowing for a more concise encoding.

4 An Algorithm Using Set Reconciliation

We now present our new algorithm based on set reconcilication. We limit ourselves to a single round of messages between client and server, and measure the communication cost in terms of the total number of bits exchanged between the parties. We assume that the client has knowledge of (an upper bound on) the size of the symmetric difference between the sets of hashes on both sides; we address this assumption in the next section.

The main idea of our algorithm is as follows: We locally partition both versions of the file into overlapping blocks using the *2-way min* technique, and represent the blocks by their hashes. We then use a set reconciliation protocol consisting of a single message from client to server, such that the server knows

[1]We note that a claim in [12] that the winnowing method in [26] does not improve on the older method is incorrect and due to an implementation bug.

which of the blocks in f_{new} are already known to the client. Then the server transmits f_{new} to the client in two parts: Blocks not known to the client are encoded using a compression algorithm similar to *gzip*, while the information about the ordering of blocks within the new file is communicated in an optimized manner that exploits the fact that for each block there is usually only a very small number (often just one) of other blocks that can follow this block (i.e., that start with exactly the right characters). Here are the details:

0. At both server and client:

(a) Use *2-way min* to partition the local file into a number of blocks, and compute a hash for each block. Let S_c and S_s be the sets of hashes at the client and server, respectively.

1. At the client:

(a) Let d be the symmetric difference between the two sets of hashes. Use the set reconciliation algorithms described in [18] to evaluate the characteristic polynomial on S_c on d randomly selected points, and transmit the results to the server.

2. At the server:

(a) Use the d evaluations to calculate the symmetric difference between S_c and S_s, i.e., the hashes in $S_c - S_s$ and $S_s - S_c$.

(b) The server goes through f_{new} to identify all blocks that are not known by the client. Any two consecutive blocks not known to the client are merged into one.

(c) The server now sends to the client the following information in suitably encoded form:

(i) the number of blocks in f_{new},

(ii) a bit vector specifying which of the hashes of f_{old} (sorted by value) also exist in f_{new},

(iii) a bit vector specifying which of the blocks in f_{new} (sorted by position) also exist in f_{old},

(iv) the lengths of all blocks in f_{new} that are not in f_{old},

(v) the interiors of these blocks themselves in suitably coded form, and

(vi) an encoding of the sequence of matched blocks in f_{new}.

Recall that in the case of fixed block size, we need hash values of about $\lg(n) + \lg(n/b) + k$ bits ([12]) in order to keep the chance of a collision below $1/2^k$, where n is the file size and b is the (average) block size. However, in the case of a content-dependent partitioning such as *2-way min* we only need $2\lg(n/b) + k$ bits, since each hash is only compared to n/b blocks on the other side. Thus, we use $2\lg(n/b) + k$ bits per hash in the set reconciliation protocol. Because fewer bits per hash are needed than in the fixed block-size case, this in turn allows profitable use of smaller block sizes. Moreover, the significant overlap between blocks makes it highly efficient to code the ordering of the blocks, since each block can only be followed by another block that starts with the last c characters of the previous block. Overlapping blocks also assure that only the interior of unmatched blocks must be sent.

4.1 Details on Encoding and Decoding

Now a few more details on the precise encoding of the various items. The two bit vectors sent from server to client can be compressed using arithmetic coding, though they are fairly compact already. The lengths of the unmatched blocks in f_{new} are Golomb-coded [35], while the interior parts of the unmatched blocks are compressed using a *gzip*-like algorithm similar to the one employed in *rsync*. The sequence of matched blocks is coded as follows: If the previous block was not matched, then $\lg(x)$ bits are used to specify the next block, where x is the total number of matched blocks. Else $\lg(y)$ bits are used, where y is the number of possible blocks that can follow the previous one. In many cases, $y = 1$, and there is a unique continuation.

Finally, the client reconstructs f_{new} by determining which blocks exist in both files, and determining for each such block which other such blocks can follow it. Then the new blocks in f_{new} unknown to the client are decoded, and finally the matched and new blocks are assembled into f_{new} based on the other information received.

4.2 A Variant Based on Golomb Coding

Note that set reconciliation is used in our algorithm simply as an efficient way to transmit a set of hashes from client to server, and that we could easily replace it by another method without changing anything else. In particular, we propose as an alternative to simply sort all the client hashes and then send them to the server by Golomb-coding (see [35]) the gaps between hash values, resulting in additional compression. As we will see later, set reconciliation is a good method for transmitting the hashes when the two files are very similar, but not suitable for files with significant differences.

4.3 Comparison to Previous Work

While our approach is based on very similar ideas as the algorithm in [1], there are several important differences. First, our method uses a single round, compared to two rounds used in [1]. Second, we use content-dependent partitioning techniques, instead of fixed masks, to partition files into blocks. This has significant benefits since it means that every edit operation (difference) between the two files can only impact a constant number (often only one) of blocks and block hashes. In contrast, in [1] the number of impacted blocks is proportional to the mask size, which is usually chosen as $\lg n$. As we discuss later, this in fact results in an asymptotic difference under certain assumptions. Third, the overall structure of our method is quite similar to that of *rsync*. This allows an implementation similar to *rsync* as a set of simple stateless communicating processes. In fact, our method can be seen as a unified approach that combines *rsync* and the approach in [1].

There are also several more minor differences. Because of the similarity to *rsync*, we have the choice of using reconciliation or Golomb coding for the hashes, depending on the characteristics of the data. Our method also includes compression for unmatched literals; in our experience this makes a very significant difference in practice on many data sets. Finally, we use a simpler approach for specifying and coding the sequence of blocks (i.e., the Eulerian path in [1]) whose running time does not depend significantly on the number of paths in the graph. Note that the actual number of round-trips of course depends on the set reconciliation protocol that is used. A recent recursive protocol in [17] gives a significant reduction in computational cost at the receiver at the cost of additional communication rounds, when compared to the basic single-message approach in [18].

4.4 Communication Complexity

We now discuss the asymptotic communication complexity of our algorithm and compare it to that of other approaches. In the discussion, we assume two files f_{new} and f_{old} of length n each with edit distance k, where the allowed edit operations are change, insertion, and deletion of single characters and moves of blocks of characters. (More general edit distance measures also allow copies and deletions of blocks, but these are tricky to analyze as distances between files are not symmetric.) There is a well known lower bound of $\Omega(k \lg n)$ bits of communication in the worst case, and a matching upper bound was established in [21]. However, that algorithm cannot be implemented efficiently in practice as it requires the inversion of certain hash functions at the receiver. The practical multi-round algorithms for synchronization in [27, 16, 22, 6, 30] achieve a communication complexity of $O(k \lg n \lg(n/k))$ bits using $\lg(n/k)$ communication rounds. A recent result in [12] matches this bound with a single round. Thus, there remains a logarithmic gap between the lower bound and the best practical protocols.

The analysis of the algorithm based on set reconciliation in [1] implies a bound of $O(k \lg^2 n)$ bits with two round-trips for average-case files, i.e., the case where one file is created at random while the other file is any file of edit distance at most k. (This is for the case of a mask length of $\Theta(\lg n)$.) Our algorithm, in contrast, can be shown to achieve a bound of $O(k \lg n)$ bits in a single round on such average-case files; this is due to our use of content-dependent partitioning techniques to define overlapping blocks – as a result each edit operation only affects a constant number of blocks. We note that this type of average-case analysis is somewhat problematic as many string processing problems are much easier on randomly generated strings (which are unlikely to have any large repeated substrings). Thus, we do not give further details here as this result is only of limited theoretical interest. A practical algorithm with the same bound in the worst case, on the other hand, would be a major result.

4.5 Preliminary Experimental Results

We now present some preliminary experimental results that show the potential of the new algorithm. For the experiments, we used the *gcc* and *emacs* data sets also used in [30, 12, 11], consisting of versions 2.7.0 and 2.7.1 of *gcc* and versions 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. We also ran additional experiments on the $html$ data sets that were used in [30], consisting of a set of ten thousand pages crawled randomly from the web, and the same pages recrawled two days later. Each set of pages has a total size of around 140MB, with about 14KB per page on average. Some of the files are not updated at all between crawls, while others change only slightly. For each data set, we measured the cost of updating all files in the older version to the newer one. We assume that only files that have changed are updated, while unchanged files are detected through an MD5 hash on the entire file that is exchanged before the algorithm, at a cost of 16 bytes per file.

In Figure 4.1 and 4.2, we compare the performance of the following five approaches on two data sets, gcc and $html$: The algorithm based on set-reconciliation, the variant based on Golomb-coded hashes in Subsection 4.2, a combination of these two that magically always chooses the best of the two depending on the data, *rsync*, and an optimization of *rsync* from [12] called FBS which was shown to always (at least slightly) outperform *rsync* through various minor improvements. For each algorithm, we did multiple runs with different block sizes, and plot the average resulting block size versus the total communication cost.

We see that for both *gcc* and *html*, the approach based on reconciliation does significantly better than the Golomb-based

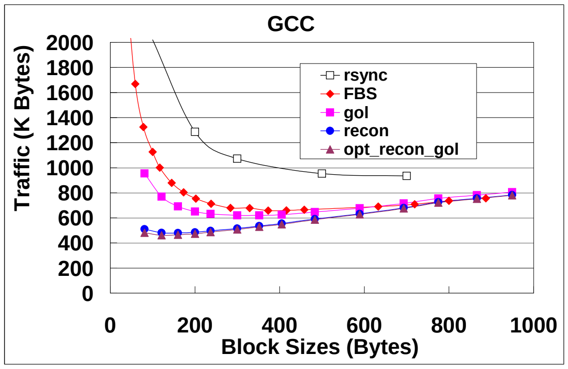

Figure 4.1. Comparison of algorithms on the *gcc* data set. The graphs from top (worst) to bottom are rsync, FBS, Golomb, reconciliation, and the optimal combination of the last two.

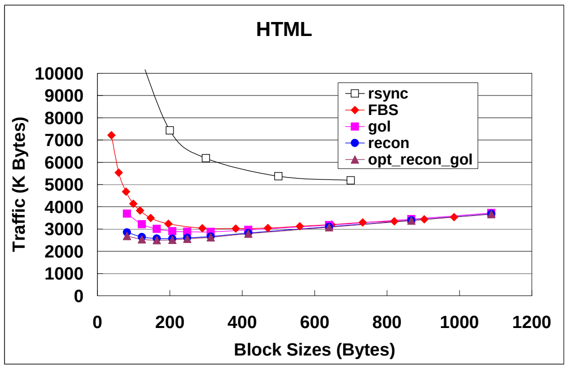

Figure 4.2. Comparison of algorithms on the *html* data set. The graphs from top (worst) to bottom are rsync, FBS, Golomb, reconciliation, and the optimal combination of the last two.

approach, which itself outperforms FBS. Moreover, the new methods achieve their optimum at smaller block sizes than FBS, as they communicate fewer bits per hash. In particular, Golomb saves over FBS both by using fewer bits per hash, and by sorting and Golomb-coding the hashes before transmission. However, set reconciliation is better than Golomb on almost all files, and thus the approach that chooses the best of the two does only marginally better than always using set reconciliation. In total, the best approach for *gcc* needs about 460 KB, compared to about 760 KB for FBS; and the best approach for *html* needs about 2500 KB, compared to about 3018 KB for FBS. Thus, the experiments show that our basic algorithm performs very well on fairly similar data sets.

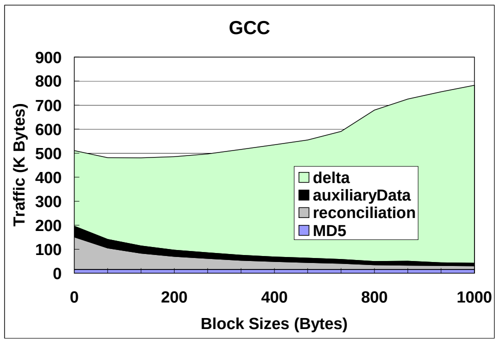

Figure 4.3. Costs of the different data structures transmitted in the reconciliation-based protocol, for *gcc*.

Next, in Figure 4.3, we look at the various contributions to the overall transmission cost, for the *gcc* data. Most of the cost is due to the transmission of blocks in f_{new} that do not appear in f_{old} (called *delta* in the figure). This cost of course increases with the block size. On the other hand, the cost of sending the evaluations in the actual set reconciliation operation increases for smaller blocks. The other data structures sent from server to client, i.e., the bit vectors, block lengths, and the encoding of the Eulerian path segments, make up only a small part of the overall cost (labeled as *auxiliary*), while the cost of the MD5 hash of each file is negligible.

4.6 Effect of Similarity on Performance

We expect the reconciliation-based algorithm to significantly outperform other methods on fairly similar files, but less on files with more significant changes. We verified this conjecture by creating artificial data sets with varying degrees of similarity, by *morphing* the *gcc* data with other unrelated data through a simple Markovian copy process. The results, shown in Figure 4.4, clearly show that the relative advantage of reconciliation over Golomb coding is most significant when the files are fairly similar.

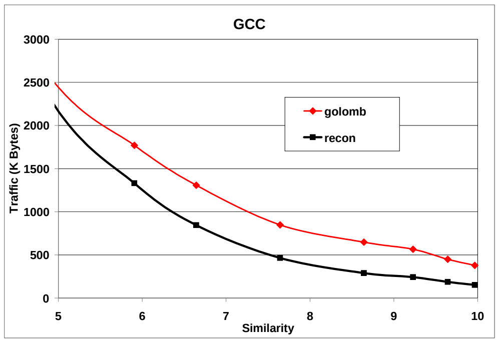

Figure 4.4. Comparison of Golomb and reconciliation methods for data with varying degrees of similarity. A value of k on the x-axis means that about a $1/2^k$ fraction of the content of each file has been changed (where each changed region has an expected length of 5 characters).

However, the situation is different on data sets with fairly different files, e.g., *emacs*, where the two versions are substantially less similar than for *gcc* or *html*. In this case, shown in Figure 4.5, the FBS algorithm wins out against both Golomb and reconciliation, and even does slightly better than the optimal combination of the other two for the best block size (about 3120 KB vs. 3200 KB). Note that in the case of *emacs*, fairly small block sizes are best for all algorithms, since larger block sizes are not sufficient for identifying the limited amount of redundancy between the two sets. Also, Golomb is on average slightly better than reconciliation (as expected from the previous figure).

To summarize, the results indicate that the new approach has significant potential for collections with fairly similar versions. However, to make the algorithm practical, we still need to find a way to estimate the symmetric difference, and ideally also to adaptively choose either the Golomb or set reconciliation strategy based on the underlying data.

5 Integrating Sampling into the Algorithm

Recall that in the previous section, we assumed that the client knows a good upper bound on the size of the symmetric

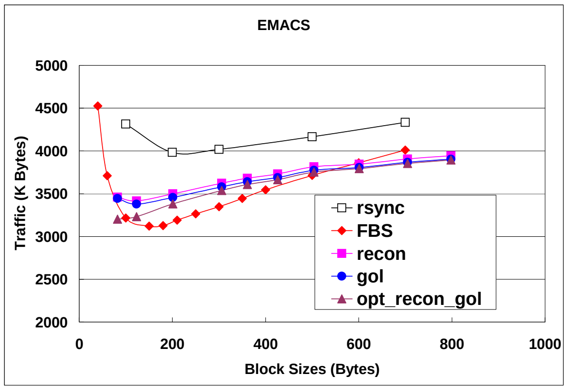

Figure 4.5. Comparison of different algorithms on *emacs* data. The graphs from top (worst) to bottom are for rsync, reconciliation, Golomb, the optimal combination, and FBS.

difference between the two sets of hashes. However, such information is usually not available in practice. In order to overcome this problem, we propose to use random sampling to estimate the symmetric difference. This can be done during the first exchange of meta data in the *rsync* framework, and it also allows us to choose either set reconciliation or Golomb coding depending on the degree of similarity between the files.

5.1 A More Practical Algorithm

Our goal is to estimate the symmetric difference d between the sets of hashes of f_{old} and f_{new} by exchanging suitable samples before running the algorithm. We note that this can be done without increasing the number of round-trips within an *rsync*-like structure, by sending samples from server to client (instead of from client to server) during the first exchange of meta data (such as MD5 and directory info). This allows the client to estimate the number of evaluations that have to be sent to the server, at some cost in bandwidth.

Our goal is not merely to estimate the difference d, but to provide an upper bound that is correct with fairly high probability. If our estimate for d is too low, then the server will be unable to derive the set difference from the d evaluation points sent by the client; in this case we assume that the server sends the entire f_{new} in compressed form resulting in a significant extra cost. Thus, we cannot use a “best estimate” for d but need to pad this such that the probability of an underestimate is very small, say less than 0.1%. The important point here is that a larger sample size can decrease costs by allowing us to provide a much tighter upper bound for d than a very small sample.

We choose as our sample the subset of those hashes that are $x \bmod 2^y$ for some x and y. We can decrease the transmission cost of the sample by sending fewer bits per sampled hash. In particular, we do not have to send the y bits used to select the sample (since the other side will only compare the sample to its own hashes $x \bmod 2^y$), and we can further reduce the cost of each hash by removing additional bits and later correcting for the expected number of resulting collisions. (Note that this is somewhat similar to using a compressed Bloom filter for this purpose.)

In the end, we found that using about $\lg(S) + 8$ bits per sample element performed best, where S is the number of samples. The samples were then sorted and again Golomb coded, for a net cost of slightly more than 9 bits per sample element. Sample size was chosen based on the number of blocks in the file. For files of size less than a few KB, no sampling was done, and the

Golomb-based algorithm was used instead of set reconciliation. Since samples are sent together with the MD5 and directory information, they are sent even for those files that were completely unchanged. (We also experimented with an alternative technique for estimate the difference between the sets based on a reduction to Hamming distance as described in [6], but the observed differences were small and thus we omit the results from this paper.)

Once we have an estimate for d, we can also use this in order to decide whether to use reconciliation or Golomb coding to transmit the hashes. While such a hybrid approach did not seem to give much benefit in the ideal case considered in the previous section, it is actually more useful now. The reason is that the required padding for d increases the cost of the reconciliation-based approach, and sometimes it is now better to use Golomb coding (which does not rely on d) where with exact knowledge of d we would choose reconciliation.

In Figure 5.1, we compare the following four methods on the *gcc* data sets: (1) FBS with no sampling, (2) the idealized algorithm from the previous section that "knows" d without sampling and chooses the best of Golomb and set reconciliation, (3) an algorithm that tries to make this choice based on sampling, and (4) an algorithm that always uses set reconciliation (except for very small files) and that uses sampling only to select d. Both sampling-based methods do significantly better than FBS, though not quite as well as the idealized algorithm due to the overheads of sampling and padding. Overall, the best method with sampling achieves about 533 KB, versus 760 KB for FBS.

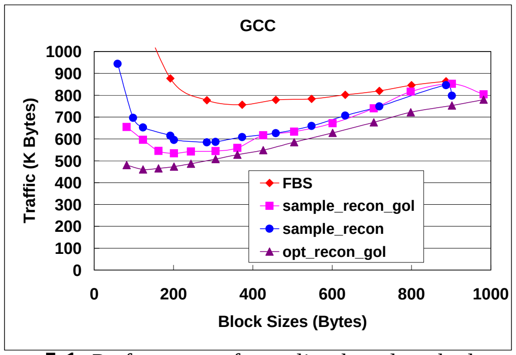

Figure 5.1. Performance of sampling-based methods on *gcc*.

In Figure 5.2, we see the contributions of the various data structures to the overall transmission cost for three block sizes and three of the methods (excluding FBS, whose internals we have not introduced here). We note that the cost of the sampling itself is fairly low, while the cost due to padding is more significant. Note that in the two methods that can choose from either Golomb or reconciliation, the cost of the evaluations for the best estimates for d (i.e., excluding padding) is very small[2] and hardly visible (the second color from the top in the figure).

For *emacs*, we observed that the method that uses sampling plus the best of Golomb or reconciliation does somewhat worse than FBS (3361 KB vs. 3120 KB for FBS), as expected. The figure is omitted due to space constraints.

5.2 A Hybrid Method with FBS

From the above experiments, we can see that FBS sometimes performs better than methods based on variable block sizes on

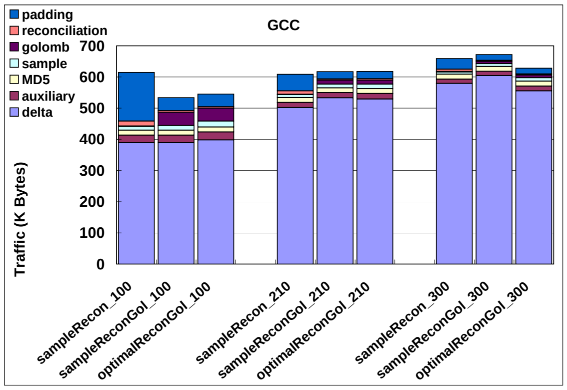

Figure 5.2. Costs of various parts of the three algorithms on *gcc*, for w values of 100, 210, and 300, with resulting average block sizes of 199, 405, and 589, respectively. (Colors in the charts and on the right are in the same order.)

fairly different files. To address this problem, we investigated a hybrid method that combines set reconciliation, Golomb, and FBS. In particular, using our sampling-based estimate of the similarity between the two files, the new hybrid applies FBS on files with similarity below some threshold, and chooses between Golomb and reconciliation otherwise. In Figure 5.3, we compare this new hybrid to the previous methods as well as to an idealized version of the new hybrid that always chooses the best of the methods without sampling.

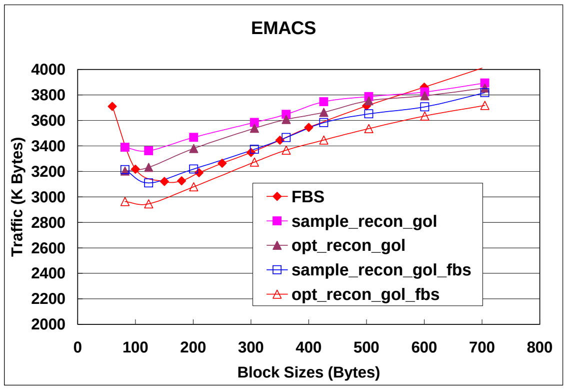

Figure 5.3. A hybrid algorithms with FBS on *emacs* data.

From Figure 5.3, we observe that the new hybrid algorithm with sampling does only very slightly better than FBS, while the idealized version of it does significantly better. On the other hand, when we ran experiments on *gcc*, adding FBS into the hybrid made almost no difference at all both in the idealistic and sampling-based case; this is because in *gcc* most files are very similar and thus FBS is almost never a good choice. However, this does not mean that the new hybrid is not useful. In fact, the point here is that this hybrid performs well over many different types of data sets, matching the best method in each particular case. Also, results for the idealized case on *emacs* indicate that additional benefits could be achieved through improved sampling techniques.

5.3 Estimating a Good Block Size

One problem with single-round synchronization algorithms is that performance depends on the choice of the block size. It would be nice if we could also use our sample in order to select a good block size; unfortunately this sounds easier than it

[2]This is not the same as the cost for reconciliation in the ideal method in Figure 4.3, since we are more likely to choose reconciliation over Golomb in cases where we underestimate the difference between the hash sets.

is. We experimented with several approaches for this. There are two main problems: First, we may need to sample hashes for several block sizes in order to select the best one, increasing the sample size. Second, to choose a good block size, it does not suffice to estimate the number of blocks that are in common. Instead, we also need to know the compressibility of the unmatched parts of f_{new}, since there is a trade-off between the size and number of blocks and the size of the unmatched parts of the file. Our results indicate that this compressibility ranges widely within our data sets, from a factor of 2 to more than 10, which means that simply dividing the size of the unmatched parts by an "average compression factor" of, say, 4 for *gzip* methods does not work. Note that this problem of choosing a good block size is not unique to our new methods, but also exists in *rsync* and all previous single-round methods.

6 Concluding Remarks

In this paper, we have described and evaluated new algorithms for remote file synchronization that use a single communication round. We also proposed and explored the use of random sampling techniques for further optimizations. Our results show that significant improvements in communication costs can be achieved in many cases, assuming the files are quite similar.

We believe that the lessons learned from this work are as follows: First, while earlier approaches [12] based on content-dependent block partitioning did not do as well as the fixed-block partitioning in *rsync*, the newer partitioning techniques in [26, 31] do much better and provide an interesting alternative to fixed-size blocks. Second, using set reconciliation techniques with overlapping content-dependent partitioning is a promising approach that does well on collections where versions are quite similar. Third, sampling techniques can efficiently decide whether set reconciliation should be used and how many evaluations to send, but it is not easy to use sampling to decide on the best block size, due to the significant variations in the compressiblity of the unmatched blocks.

References

[1] S. Agarwal, V. Chauhan, and A. Trachtenberg. Bandwidth efficient string reconciliation using puzzles. *IEEE Transactions on Parallel and Distributed Systems*, 17(11):1217–1225, November 2006.

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

[3] N. Bjorner, A. Blass, and Y. Gurevich. Content-dependent chunking for diffential compression: The local maximum approach. MSR-TR-2007-102, Microsoft, 2007.

[4] A. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences (SEQUENCES'97)*, pages 21–29. IEEE Computer Society, 1997.

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

[6] 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.

[7] 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.

[8] P. Eaton, E. Ong, and J. Kubiatowicz. Improving bandwidth efficiency of peer-to-peer storage. In *Proc. of the 4th IEEE Int. Conf. on Peer-to-Peer Computing*, pages 80–89, Aug 2004.

[9] 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.

[10] T.H. Haveliwala, A. Gionis, and P. Indyk. Scalable techniques for clustering the web. In *Proc. of the WebDB Workshop*, 2000.

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

[12] U. Irmak, S. Mihaylov, and T. Suel. Improved single-round protocols for remote file synchronization. In *Proc. of the IEEE INFOCOM Conference*, March 2005.

[13] U. Irmak and T. Suel. Hierarchical substring caching for efficient content distribution to low-bandwidth clients. In *Proc. of the 14th Int. World Wide Web Conference*, May 2005.

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

[15] P. Kulkarni, F. Douglis, J. LaVoie, and J. Tracey. Redundancy elimination within large collections of files. In *USENIX Annual Technical Conference*, 2004.

[16] J. Langford. Multiround rsync. January 2001. Unpub. manuscript.

[17] Y. Minsky and A. Trachtenberg. Practical set reconciliation. Boston University Technical Report, 2002.

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

[19] 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.

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

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

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

[23] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *First USENIX conference on File and Storage Technologies*, 2002.

[24] 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.

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

[26] 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.

[27] 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.

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

[29] D. Starobinski, A. Trachtenberg, and S. Agarwal. Efficient PDA synchronization. *ACM Transactions on Mobile Computing*, 2003.

[30] 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.

[31] D. Teodosiu, N. Bjorner, Y. Gurevich, M. Manasse, and J. Porkka. Optimizing file replication over limited bandwidth networks using remote differential compression. MSR-TR-2006-157, Microsoft, 2006.

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

[33] U.Manber. Finding similar files in a large file system. In *Proc. of the 1994 USENIX Conference*, May 1994.

[34] Author Unspecified. How rsync works – a practical overview. `http://samba.anu.edu.au/rsync/how-rsync-works.html`.

[35] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, second edition, 1999.