Remote File Synchronization Single-Round Algorithms

Deepak Gupta
Bhagwan Parshuram Institute of Technology
Delhi, India

Kalpana Sagar
University School of Information technology
Delhi, India

ABSTRACT

Remote file synchronization has been studied extensively over the last decade, and the existing approaches can be divided into single-round and multi-round protocols. Single-round protocols are preferable in scenarios involving small files and large network latencies (e.g., web access over slow links) due protocol complexity and computing and I/O overheads. The best-known algorithms which are used for synchronization of file systems across machines are rsync, set reconciliation, Remote Differential Compression & RSYNC based on erasure codes.

In this paper we will discuss the remote file synchronization protocols and compare the performance of all these protocols on different data sets.

Index Terms *— Remote files synchronization (RSYNC), Remote Differential Compression (RDC), Set Reconciliation (Recon), GCC, HTML, EMACS.*

I. INTRODUCTION

Remote file synchronization has been studied extensively over the last decade, and the existing approaches can be divided into single-round and multi-round protocols. Single-round protocols 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. (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 expounded. Experiments have shown that multi-round protocols can provide significant bandwidth savings over single-round protocols on typical data sets. However, multi-round protocols have several disadvantages in terms of protocol complexity, 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.

For instance consider the case of a group of people collaborating over email to produce a large PowerPoint presentation, sending it back and forth as an attachment each time they make changes. An analysis of typical incremental changes shows that very often just a small fraction of the file changes. Therefore, a dramatic reduction in bandwidth can be achieved if just the changes are communicated across the network. A change affecting 16KB in a 3.5MB file requires about 3s to transmit over a 56Kbps modem, compared to 10 minutes for a full transfer.

Imagine that you have two files, A and B, and you wish to update B to be the same as A. The obvious method is to copy A onto B. Now assume that the two files are on machines connected by a slow communications link, for example a dial up IP link. If A is large, copying A onto B will be slow. To make it faster you could compress A before sending it, but that will usually only gain a factor of 2 to 4.

1.1 The setup for the file synchronization problem

Fig 1 shows the general setup for remote file synchronization.

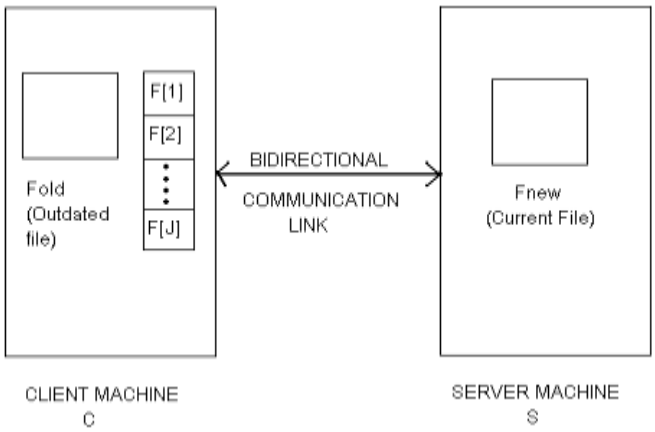

Fig 1 General Setup for remote file synchronization

Fig 1 shows that, we have two files (strings) fnew, fold Є Σ* 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 fold as the outdated file and to fnew as the current file. We assume that C only has a copy of fold and S only has a copy of fnew. Our goal is to design a protocol between the two parties that results in C holding a copy of fnew, 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, $0 \le i < |f|$, 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 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. The above said problem is also discussed in [2, 7, 15, 16].

The rest of this paper is structured as follows: Section II summarizes the basic algorithms used by RSYNC, ERASURE CODE, SET RECONSILIATION and RDC protocols, Section III gives the experimental comparison between all of the above protocols on different data sets. Finally the paper concludes in section IV.

II. TECHNICAL PRELIMINARIES

In this section, we will describe different remote file synchronization protocols along with their approach used to synchronize two files which are placed on two different machines using a communication medium.

A. The RSYNC Algorithm

The basic idea in rsync as discussed in [2] 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.

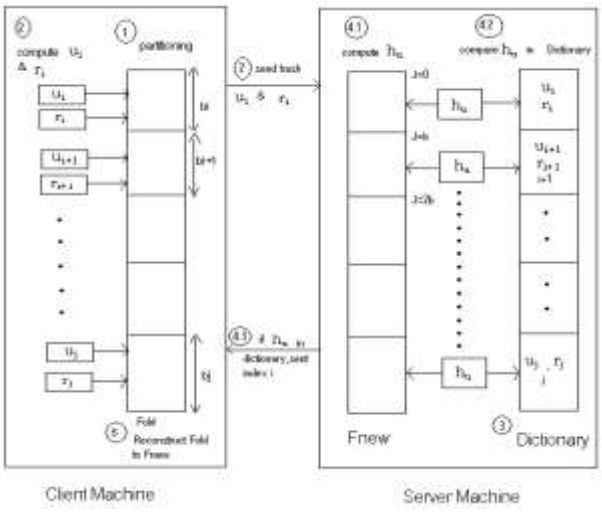

Fig 2.1 RSYNC

In rsync as shown in fig 2.1, 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 fnew for a possible match with a block in fold. The complete algorithm is as follows:

1. at the client:	
Step 1:	Partition fold into blocks Bi = fold [ib, (i + 1) b − 1] of some block size b.
Step 2:	For each block Bi, compute two hashes, ui = hu (Bi) and ri = hr (Bi), and communicate them to the server. Here, hu is a heuristic but fast hash function, and hr is a reliable but expensive hash.

2. at the server:	
Step 3:	For each pair of received hashes (ui, ri), insert an entry (ui, ri, i) into a dictionary, using ui as key.
Step 4:	Perform a pass through fnew, starting at position j = 0, and involving the following four steps:
4.1:	Compute the unreliable hash hu(fnew[j, j+b−1]) on the block starting at j.
4.2:	Check the dictionary for any block with matching unreliable hash.
4.3:	If found, and if the reliable hashes match, transmit the index i of the matching block in fold to the client, advance j by b positions, and continue.
4.4:	If none found, or if the reliable hash did not match, transmit symbol fnew[j] to the client, advance j by one position, and continue.

3. at the client:	
Step 5:	Use the incoming stream of symbols and indices of hashes in fold to reconstruct fnew.

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. The File Synhronization based on Erasure Codes

The basic idea underlying this approach as studied in [7] is quite simple: essentially, erasure codes are used to convert certain multi-round protocols into single-round protocols with similar communication cost. 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. To summarize, the algorithm works as follows:

Step 1:	The server partitions fnew recursively into blocks from size bmax down to bmin, and for each level computes all block hashes.
Step 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.
Step 3:	In one message, the servers send all hashes at the highest level to the client, plus the 2k erasure hashes for each level.
Step 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.
Step 5:	At the bottom level with block size bmin, 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 bmax $= \lfloor n/k \rfloor_2$, bmin = lg (n), and hashes of size 4 lg n bits.

While the protocol above 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 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 the following description note that the first three steps are identical to the previous algorithm while the roles of client and server exchanged.

Step 1:	The client partitions fold recursively into blocks from size bmax down to bmin, and for each level computes all block hashes.
Step 2:	The client applies a systematic erasure code to each level i of hashes except the top level, and computes mi erasure hashes for each level, for some appropriate mi discussed later.
Step 3:	In one message, the client sends all hashes at the highest level to the server, plus the mi erasure hashes for each level i.
Step 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 mi, the hash function is applied to all blocks that do have a matched ancestor, and the mi erasure hashes are used to recover the hashes of the other blocks. Otherwise, we stop at the previous level of hashes.
Step 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.

C. The Set Reconciliation

We now discuss the set reconciliation problem which is discussed in [15] and its relation to file synchronization. In the set reconciliation problem, we have two machines A and B that each holds a set of integer values SA and SB 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 (SA – SB) U (SB - SA). A protocol based on characteristic polynomials achieves this with a single message. We define the characteristic polynomial $\chi s(z)$ of a set S = {x1, x2… xn} as the univariate polynomial

$$\chi s(z) = (z - x_1)(z - x_2) \ldots (z - x_n) \quad \ldots \quad (1)$$

An important property of the characteristic polynomial is that it allows us to cancel out all terms corresponding to elements in SA ∩ SB, by considering the ratio between the characteristic polynomial SA and SB.

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

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, 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 $\chi_A(z) / \chi_B(z)$. Thus, if the difference between the two sets is small, then only a small amount of data has to be communicated.

Recent work 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.

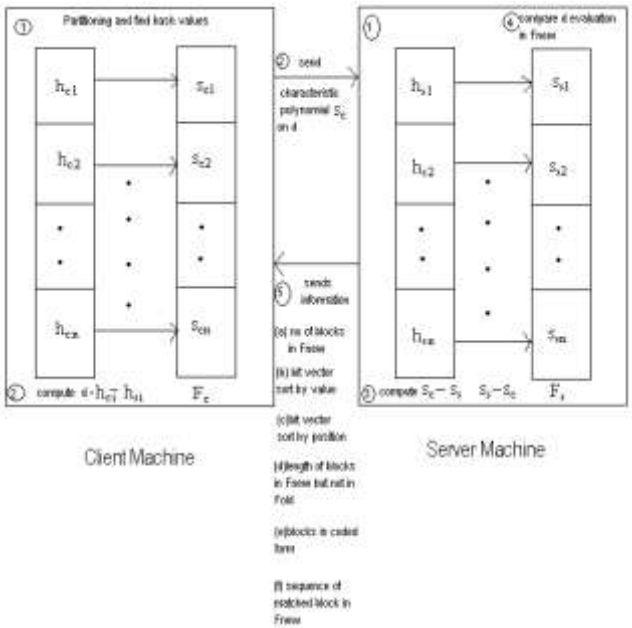

Fig 2.2 Set Reconciliation

The main idea of our algorithm as shown in fig 2.2 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 which of the blocks in fnew are already known to the client. Then the server transmits fnew 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:	
Step 1:	Use 2-way min to partition the local file into a number of blocks, and compute a hash for each block. Let SC and SS be the sets of hashes at the client and server, respectively.

1. at the client:	
Step 2:	Let d be the symmetric difference between the two sets of hashes. Use the set reconciliation algorithms to evaluate the characteristic polynomial SC on d randomly selected points, and transmit the results to the server.

2. at the server:	
Step 3:	Use the d devaluations to calculate the symmetric difference between SC and SS i.e., the hashes in SC – SS sand SS – SC.
Step 4:	The server goes through fnew to identify all blocks that are not known by the client. Any two consecutive blocks not known to the client are merged into one.
Step 5:	The server now sends to the client the following information in suitably encoded form:
5.1:	The number of blocks in fnew
5.2:	A bit vector specifying which of the hashes of fold (sorted by value) also exist in fnew
5.3:	A bit vector specifying which of the blocks in fnew (sorted by position) also exist in fold,
5.4:	The lengths of all blocks in fnew that are not in fold,
5.5:	The interiors of these blocks themselves in suitably coded form, and
5.6:	An encoding of the sequence of matched blocks in fnew

D. The Remote Differential Compression (RDC)

For completeness, we summarize the basic RDC protocol discussed in [16] used in LBFS [14]. While LBFS uses the entire client file system as a seed for differential transfers, we shall assume without loss of generality the existence of a single seed file FC, as this shall facilitate the presentation of our approach in the following sections.

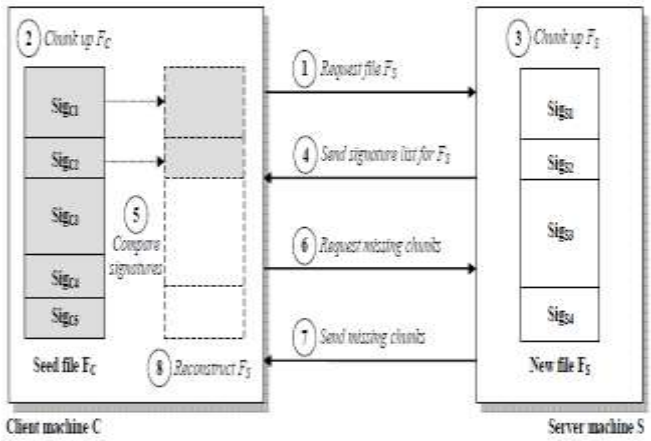

Fig 2.3 Remote Differential Compression

The basic RDC protocol assumes that the file FS on the server machine S needs to be transferred to the client machine C using the seed file FC stored on the client. FS is a new version containing incremental edits over the seed file FC. The transfer of FS from S to C is performed as follows:

Step 1:	C sends S a request to transfer files FS.
Step 2:	C partitions FC into chunks by using a fingerprinting function that is computed at every byte position of FC. A chunk boundary is determined in a data-dependent fashion at positions for which the fingerprinting function satisfies a certain condition. Next, a signature SigCk is computed for each chunk k of FC. A cryptographically secure hash function (SHA-1) is used in LBFS, but any other collision resistant hash function may be used instead.
Step 3:	Using the same partitioning algorithm as in Step 2, S independently partitions FS into chunks and computes the chunk signatures SigSj. Steps 2 and 3 may run in parallel.
Step 4:	S sends the ordered list of chunk signatures and lengths ((SigS1, LenS1)… (SigSn, LenSn)) to C. Note that this implicitly encodes the offsets of the chunks in FS
Step 5:	As this information is received, C compares the received signatures against its own set of signatures {SigC1… SigCm} computed in Step 2. C records every distinct signature value received that does not match one of its own signatures SigCk.
Step 6:	C sends a request to S for all the chunks for which there was no matching signature. The chunks are requested by their offset and length in FS.
Step 7:	S sends the content of the requested chunks to C.
Step 8:	C reconstructs FS by using the chunks received in Step 7, as well as its own chunks of FC that in Step 5 matched signatures sent by S.

In LBFS, the entire client file system acts as the seed file FC. This requires maintaining a mapping from chunk signatures to actual file chunks on disk to perform the comparison in Step 5. For a large number of files this map may not fit in memory and may require expensive updates on disk for any changes to the local file system. In our approach the seed is made up of a small set of similar files from the client file system, and can be efficiently computed at the beginning of a transfer based on a data structure that fits in memory.

The various protocols discussed in this section are shown in the table 1

Protocol Name	**Basic Technique**
RSYNC	*Partitioning in blocks and compute ui and ri hashes using hu*
RSYNC based on Erasure Codes	*Partition using bmax and bmin and compute hashes in top-down manner*
SET RECONCILIATION	*Compute hashes and characteristic polynomial*
Remote Differential Compression (RDC)	*Partitioning file into chunks and compute signatures*

Table 1

III. EXPERIMENTAL RESULTS

We now present some experimental results of all the file synchronization algorithms on different data sets given below

1. gcc - The GNU Compiler Collection (usually shortened to GCC) is a compiler system produced by the GNU Project supporting various programming languages. It is used for developing software that is required to execute on a wide variety of hardware and/or operating systems.

2. html - HTML, which stands for HyperText Markup Language, is the predominant markup language for web pages. It provides a means to create structured documents by denoting structural semantics for text such as headings, paragraphs, lists etc as well as for links, quotes, and other items.

3. emacs – Emacs is a class of feature-rich text editors, usually characterized by their extensibility. Emacs has, perhaps, more editing commands compared to other editors, numbering over 1,000 commands. It also allows the user to combine these commands into macros to automate work. The original EMACS consisted of a set of Editor MACroS for the TECO editor

As discussed in [15, 16] there are the following an experimental result shown in fig 3.1, fig 3.2 & fig 3.3 for different partitioning of specified block size in bytes and calculates the total traffic in K bytes.

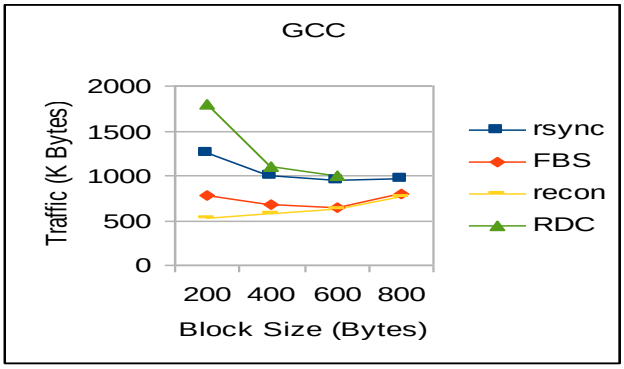

Fig 3.1 Comparison of algorithms on the *gcc* data set. The graphs from top to bottom are RDC, rsync, FBS, reconciliation

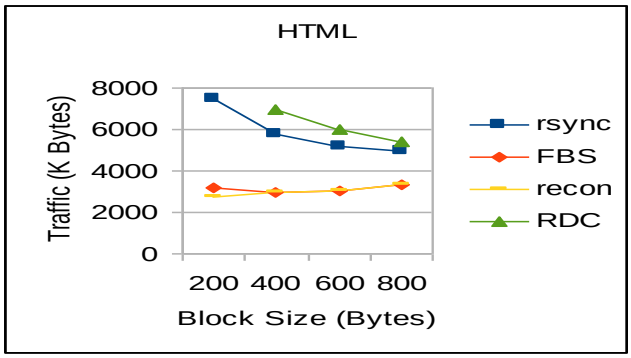

Fig 3.2 Comparison of algorithms on the html data set. The graphs from top to bottom are RDC, rsync, FBS, reconciliation

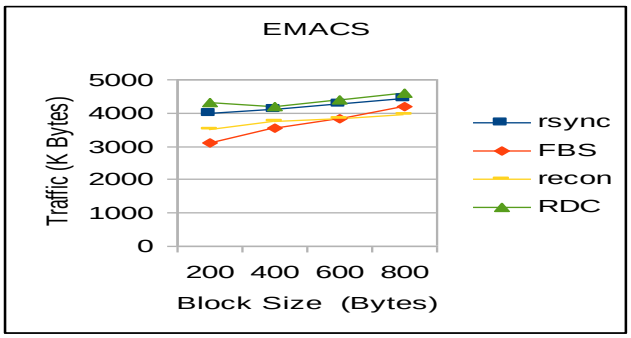

Fig 3.3 Comparison of algorithms on the emacs data set. The graphs from top to bottom are RDC, rsync, FBS, reconciliation

IV. CONCLUDING REMARK

In this paper we discussed various remote file synchronization algorithms and their performance compared to one another.
Some of the open issues that could be topics for future research in RDC (see also [16]) include determining whether an optimal chunking algorithm exists with respect to slack, and applying RDC to compressed files, other file synchronization problems is that 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 (see also [15]).

REFERENCES

[1] U. Irmak, S. Mihaylov, and T. Suel. Improved single-round protocols for remote file synchronization. In Proc. of the IEEE INFOCOM Conference, March 2005.
[2] A. Tridgell and P. Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, Australian National University, June 1996.
[3] 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.
[4] G. Cormode. Sequence Distance Embeddings. PhD thesis, University of Warwick, January 2003.
[5] 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.
[6] A. Orlitsky and K. Viswanathan. Practical algorithms for interactive communication. In IEEE Int. Symp. on Information Theory, 2001.
[7] 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.
[8] P. Noel. An efficient algorithm for file synchronization. Master's thesis, Polytechnic University, 2004.
[9] 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.
[10] A. Muthitacharoen, B. Chen, and D. Mazieres, "A Lowbandwidth Network File System," Proceedings of the 18th SOSP, Banff, Canada, 10-2001.
[11] A. Tridgell, "Efficient Algorithms for Sorting and Synchronization," PhD thesis, Australian National University, 1999.
[12] Y. Minsky, A. Trachtenberg, and R. Zippel. Set reconciliation with almost optimal communication complexity. Technical Report TR2000- 1813, Cornell University, 2000.
[13] 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.
[14] A. Muthitacharoen, B. Chen, and D. Mazieres, "A Lowbandwidth Network File System," Proceedings of the 18th SOSP, Banff, Canada, 10-2001.
[15] Hao Yan, Utku Imrak, Torsten Suel "Algorithms for Low-Latency Remote File Synchronization,".In Proc. of the IEEE INFOCOM Conference, April 2008.
[16] Dan Teodosiu, Nikolaj Bjorner, Yuri Gurevich, Mark Manasse, Joe Porkka "Optimizing File Replication over Limited-Bandwidth Networks using Remote Differential Compression,", Technical Report MSR-TR-2006-157, Microsoft Research Nov 2006
[17] I. H. Witten, A. Moffat, and T. C. Bell. Managing Gigabytes: Compressing and Indexing Documents and Images. Morgan Kaufmann, second edition, 1999.