Improved File Synchronization Techniques for Maintaining Large Replicated Collections over Slow Networks*

Torsten Suel *Patrick Noel* *Dimitre Trendafilov*

CIS Department
Polytechnic University
Brooklyn, NY 11201

suel@poly.edu, pnoel@bumblebee.poly.edu, dtrend01@utopia.poly.edu

Abstract

We study the problem of maintaining large replicated collections of files or documents in a distributed environment with limited bandwidth. This problem arises in a number of important applications, such as synchronization of data between accounts or devices, content distibution and web caching networks, web site mirroring, storage networks, and large scale web search and mining. At the core of the problem lies the following challenge, called the file synchronization problem: given two versions of a file on different machines, say an outdated and a current one, how can we update the outdated version with minimum communication cost, by exploiting the significant similarity between the versions? While a popular open source tool for this problem called rsync *is used in hundreds of thousands of installations, there have been only very few attempts to improve upon this tool in practice.*

In this paper, we propose a framework for remote file synchronization and describe several new techniques that result in significant bandwidth savings. Our focus is on applications where very large collections have to be maintained over slow connections. We show that a prototype implementation of our framework and techniques achieves significant improvements over rsync. *As an example application, we focus on the efficient synchronization of very large web page collections for the purpose of search, mining, and content distribution.*

1 Introduction

Consider the problem of maintaining large replicated collections of files, such as user files, web pages, or other documents, over a slow network. In particular, assume that we have two machines, A and B, that each hold a copy of the collection, and that files are frequently modified at one of the machines, say A. Periodically, machine B initiates 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 the files that have changed. Given the size of the collections, we are interested in performing the 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, remote backups, mirroring of large web and ftp sites, content distribution, and web search engines, 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 be able to perform the update by sending only an amount of data proportional to the 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 *file synchronization problem*. Our work is primarily motivated by several applications in large scale web search and content distribution discussed later, but our techniques are applicable to the more general case and we believe that file synchronization is a fundamental operation in distributed systems. We note that there is a very widely used open source software tool called *rsync* that addresses exactly this problem and that is described in [47, 48]. Our goal is to derive techniques that achieve significant savings over *rsync* particularly in the case of large collections and slow networks.

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 byte 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 also note that the problem is much easier if all update operations to the files are saved in an update log that can be transmitted to the other machine. However, in many cases such logs are not available. We are also 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 [3, 39]. It is left up to an application to decide when and how often files should be synchronized.

*Work supported by NSF CAREER Award NSF CCR-0093400 and by New York State through the Wireless Internet Center for Advanced Technology (WICAT) at Polytechnic University.

IEEE COMPUTER SOCIETY

1.1 Applications Motivating Our Work

The *rsync* file synchronization tool is currently widely used to exchange user files between different machines (e.g., between a machine at work and a machine at home), to mirror web and ftp sites, and to perform backup over a network. In addition to these general scenarios, we are particularly interested in the following potential applications:

1. **Sharing crawled web pages for mining and search:** One of the main bottlenecks in large-scale web search and mining is the cost of crawling large sets of web pages and then keeping these pages up to date. A lot of recent work has focused on efficient strategies for recrawling changing web pages based on their past history of updates [6, 7, 8, 9, 14]. An alternative or complementary approach would be to share the results of recrawls between many parties. For example, several organizations or even nodes in a P2P system [19] could independently perform crawls and later exchange their results, or one centralized high-performance crawler could allow clients to obtain the latest versions of web pages of interest. As an example of the latter, the Stanford WebBase project [20] enables researchers at other institutions to receive a feed of web pages from the WebBase collection. However, due to bandwidth limitations, large data sets today are still frequently shared via "sneakernet", i.e., by sending disks or tapes by mail [18]. Use of file synchronization would allow other organizations to receive updated content over the network at a fraction of the current bandwidth cost. In fact, our main motivation for this work is to build a system for efficiently sharing large recrawls over a wide area network.
2. **Maintaining massive collections at clients:** Given the speed at which hard disk capacity is currently expanding, several researchers have considered the possibility of storing and maintaining the entire web, or large parts of the world's content, at desktop machines. For example, Garcia-Molina [17] outlined a scenario where the world's changing content is distributed to end users in monthly or weekly updates shipped out on future versions of CDs or DVDs. One could also imagine instead using file synchronization to maintain large up-to-date collections at desktop clients for personalized browsing, search, or mining. Browsers such as *Internet Explorer* already allow users to *subscribe* to web pages that are then periodically downloaded and stored; the proposed techniques could be used to significantly improve the efficiency of this process.
3. **Server-Friendly Web Crawling:** Closely related to the first scenario is the idea of integrating file synchronization into web servers to support more efficient recrawling. We caution that there have been previous proposals to modify servers for this purpose [5, 16], but that widespread adoption of such schemes appears unlikely for various reasons.
4. **Caching and Content Distribution Networks:** Several companies in the CDN space have studied and deployed file synchronization techniques. We are not aware of any published work in this direction, but file synchronization techniques are a natural approach for updating content that is widely replicated at the network edge.
5. **Replication in P2P File Sharing:** While much of the content in current file sharing networks is static, file synchronization could be used to maintain dynamic content that is replicated for fault tolerance or performance.

All of these scenarios have in common that they involve massive amounts of data, and thus bandwidth efficiency is of primary importance. Our goal is to design improved protocols for file synchronization that use multiple roundtrips to significantly decrease the amount of data sent over the network.

2 File Synchronization and the *rsync* Tool

We now define the file synchronization problem and describe the algorithm of Tridgell and MacKerras [47, 48], which forms the basis of the widely used *rsync* tool.[1] We note that a similar approach was proposed by Pyne in a US Patent [38].

2.1 Problem Definition

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 assume that C only has a copy of f_{old} and S only has a copy of f_{new}, and the 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 also refer to f_{old} as the *outdated file* and to f_{new} as the *current file*. 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 until (and including) j.

2.2 The *rsync* Algorithm

The basic approach in *rsync*, as well as in our 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. Then the steps in *rsync* are as follows:

1. At the client:

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

[1] Available at `http://rsync.samba.org/`.

IEEE COMPUTER SOCIETY

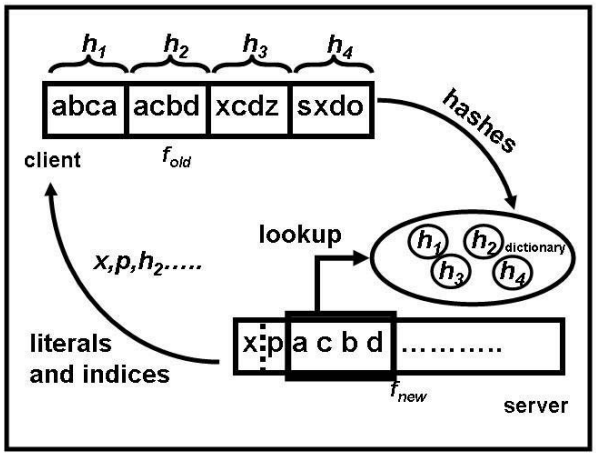

Figure 2.1. The *rsync* algorithm on a small example. The client sends a set of hashes while the server replies with a stream of literals and indices identifying hashes.

(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 function.

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 structure, using u_i as the 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 the 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 2.1. 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 can simply transfer the entire file. The reliable checksum is implemented using MD4 (128 bits), but only two bytes of the MD4 hash are used since this provides sufficient power. 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.

2.3 Discussion of *rsync* and Related Proposals

Clearly, the choice of block size is critical to the performance of the algorithm, but this choice depends on the degree of similarity between the two files – the more similar, the larger the optimal block size. 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 metrics such as edit distance [34].

In practice, *rsync* uses a default block size of 700 bytes except for very large files where a larger block size 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 are transmitted for each discovered match, while simply applying *gzip* to the unmatched blocks might result in a reduction to about 25% of the size. Other algorithms proposed in [10, 25, 34] are based on recursive splitting of unmatched blocks. We will also adopt this approach, but combine it with several other ideas that save on communication costs and allow us to utilize much smaller block sizes efficiently.

We note that recursive splitting, as some of our other techniques, increases the number of roundtrips between the two parties. However, as in *rsync* itself, the roundtrip latencies are not incurred for each file since many files can be processed simultaneously. Thus, for large collections additional roundtrips are not a problem. For small files, e.g., to fetch individual web pages, a less bandwidth efficient algorithm based on a single roundtrip would be preferable.

3 Our Contributions

We study the file synchronization problem for large collections of files and documents, and propose and evaluate new techniques that significantly improve on previous approaches in terms of bandwidth usage. Our main contributions are:

- We describe a framework for file synchronization algorithms that partitions the problem into two phases, *map construction*, where the two parties use a multiround protocol to determine the common parts of the corresponding files, and *delta compression*, where the remaining parts are encoded in relation to the common

IEEE COMPUTER SOCIETY

parts and then transmitted to the other side. The framework allows for a variety of algorithms and techniques.

- Within the framework, we describe and implement a number of known and new techniques. In particular, we use recursive partitioning as proposed in [10, 25, 34]. We introduce new techniques for extending matches via "continuation hashes" and for the optimized verification of suspected matches in the two files, plus several other optimizations. The techniques are related to classical problems in group testing and "searching with liars" (also known as Ulam's Problem), and insights from these problems may lead to additional moderate improvements in the future.
- We evaluate the framework and techniques on several data sets, including a large set of changing web pages that we recrawled daily over several weeks. The results show that our algorithm allows the maintenance of large file collections with significantly lower communication costs than the widely used *rsync* tool, and in many cases comes within 50% of the best delta compressor (which provides a reasonable lower bound in practice).

The remainder of this paper is organized as follows. In the next section, we discuss some related work. Section 5 describes the new framework and algorithmic techniques. Section 6 presents the experimental evaluation of our implementation. Finally, Section 7 discusses some limitations of our current results and open problems for future research.

4 Related Work

The most important previous work is the *rsync* file synchronization algorithm proposed by Tridgell and MacKerras [47, 48], which is the basis of the very widely used *rsync* open source tool. There are a number of theoretical studies of the file synchronization problem [10, 15, 32, 33], also called the *document exchange problem* in [10]. In particular, Orlitsky [32, 33] presents almost tight bounds for the problem with varying numbers of communication phases, under some assumptions about the assumed file distance metric. However, many of the theoretical algorithms assume that a hash function is reversed as part of the decoding operation; while this is allowable under the standard model for communication complexity [24], it makes the algorithms impossible to implement in practice. An exception are algorithms proposed in [10, 15, 25, 34] that are based on recursive partitioning of blocks similar to our implementation. These algorithms can also be shown to achieve provable bounds with respect to some common file distance measures. Some limited experimental results are given in [25, 34]. Recent work in [40] presents a version of the *rsync* algorithm that updates files *in-place* without using additional temporary space.

Delta compression is the problem of encoding one file relative to another similar file, where both files are available during the encoding. Thus, file synchronization can be seen as a distributed version of delta compression where the two files are located at different machines. Our framework reduces the file synchronization problem to delta compression; on the other hand any algorithm for file synchronization also solves the delta compression problem, though typically at significantly higher cost. Some available open source tools for delta compression are described in [23, 26, 46], and an overview of delta compression and file synchronization techniques and their applications is given in [45].

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, 27, 28, 29, 30, 36, 42]. These problems 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. The work is nonetheless related to ours in two ways. First, it addresses the problem of efficienctly identifying files that have changed in scenarios where almost all objects are unchanged; afterwards, our file synchronization techniques can be applied to update those files. We do not focus on this aspect and instead use a fingerprint for each file as this is efficient enough for our data sets. Second, some of the results [27] are also based on techniques from Group Testing, while others are based on Error Correcting Codes and probably not as useful in our context.

In addition to *rsync*, there are many other tools for synchronizing data between different machines. Some of these tools, such as Microsoft's ActiveSync, Palm's HotSync, or Puma Technologies' IntelliSync, are used to synchronize data between a desktop or online account and a mobile device. They typically transfer the entire file if a change has occurred, though for record-based data such as appointments and contacts only updated records are transmitted in most cases. Recent work in [2, 44] surveys and studies synchronization techniques for handheld devices, while [3, 39] discuss correctness issues when files are modified at several locations.

Hash-based techniques similar to *rsync* have been explored by the OS and Systems community for purposes such as compression of network traffic [43], distributed file systems [31], distributed backup [11], and web caching [41]. These techniques use string fingerprinting techniques proposed by Karp and Rabin [22] to partition a data stream into blocks in a consistent manner on both sides of a communication link, and then send hash values to encode repeated substrings.

Group Testing is a set of combinatorial problems, introduced by Dorfman [12], that deal with identifying "defective" elements in a set through a sequence of simple tests on subsets. We are not aware of previous results on the exact version of the group testing problem that arises in our work here. Group testing was used by Madej [27] to identify files that have changed. Search problems with liars were introduced by Ulam [49] and have been studied extensively over the last 50 years; see the recent survey of Pelc [37] for an overview. In particular, Pelc discusses a relationship of the problem to communication over noisy channels. On the other hand, Orlitsky and Viswanathan [35] recently established a

IEEE COMPUTER SOCIETY

relationship between Error Correcting Codes for noisy channels and one-way communication problems where a receiver has related information.

Finally, several recent studies look at the type and frequency of web page updates and propose efficient strategies for refreshing pages or other objects based on observations about their past behavior [6, 7, 8, 9, 13, 14]. These techniques are complementary to ours in the context of our web page update application. Of course, if file synchronization were to become widely deployed at web servers, then this would change the cost model assumed in current recrawling strategies.

5 Our Framework and Algorithms

We now describe our technical contributions. We first introduce our basic framework in the next subsection, and then give a detailed description of techniques for the map construction phase of our framework in Subsections 5.2 to 5.5. Finally, Subsection 5.6 summarizes the resulting protocol.

5.1 A Framework for File Synchronization

Recall that the client C has a copy of an outdated file f_{old}, and the server S has a copy of the current file f_{new}. The goal is to design a protocol between the two parties that results in C obtaining a copy of f_{new}, while minimizing the communication cost. As we saw in the description of the *rsync* algorithm, hash values can be used to identify common substrings in both files, allowing the recipient to learn about the structure of the file at the other machine.

All the algorithms in our framework consist of the following two phases:

(1) **Map construction:**: in this phase, the two parties use multiple roundtrips to create an approximate representation of the parts of the two files that are identical. In particular, the client will generate a *map* of the current file f_{new}, based on hash values sent by the server, that specifies the known and unknown parts of f_{new}. The server will meanwhile maintain a *shadow map* that keeps track of the map, i.e., which parts of f_{new} are known to the client. The goal of this phase is to minimize the size of the parts of f_{new} that are unknown to the client.

(2) **Delta compression:** in the second phase, we use delta compression to transmit the unknown parts of f_{new} to the client. In particular, the server creates a reference file f_{ref}, consisting of all parts of f_{new} that are known to the client, and a target file f_{target} of all other parts. The server then performs a delta compression of f_{target} with respect to f_{ref}, and transmits the delta to the client. The client uses its map to recreate f_{ref}, then decodes the delta into f_{target}, and merges the two files again to obtain f_{new}.

We define a map of a file $f[0, n-1]$ as an array $m[0, n-1]$ with $m[i] = f[i]$ or $m[i] =$ "?" for $i = 0, \ldots, n-1$. In other words, m is identical to f in some areas (called the *known* areas), and *unknown* in the other areas, labeled by "?".

Figure 5.1 illustrates our basic approach. Suppose that machine S contains a file $f_{new} =$ "BDAFHKZER", and C contains $f_{old} =$ "ABADFHKBCZY". Suppose S splits f_{new} into three blocks "BDA", "FHK", and "ZER", and sends a fairly strong hash for each block to C. C can now create a map m_{new} of file f_{new} that looks as follows: $m_{new} =$ "???FHK???", by finding that the hash for "FHK" matches with its own substring "FHK" in positions 4 to 6 of f_{old}. As part of the map m_{new}, B may also store where it found the match in its own file f_{old}, in this case in positions 4 to 6 (the first position is 0).

Next, C sends back "010" to S, indicating that the second hash found a match but the other two did not. Now S can build a *shadow map* for f_{new}, called $s_{new} =$ "???FHK???", that describes what C knows about f_{new}. (Note that s_{new} could be represented as a simple bit array "000111000" where "0" means that C does not know this element of f_{new}. Of course, S does not know where in f_{old} the matches were found by C.) In the next communication round C will again receive hashes, for smaller blocks obtained by splitting the unknown blocks, and will try to refine the map m_{new} so that the unknown areas become smaller and smaller, while S maintains its shadow map s_{new} of f_{new}.

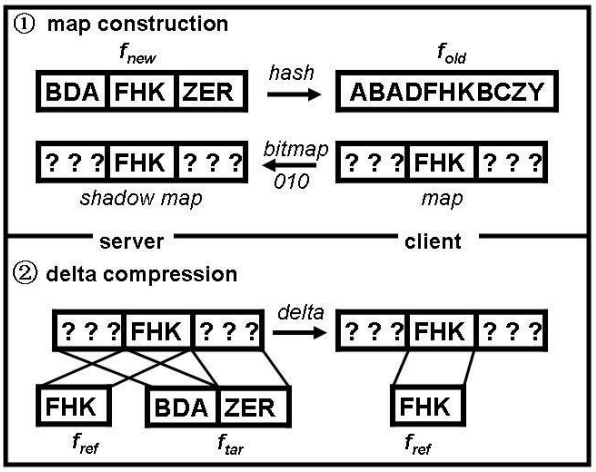

Figure 5.1. Framework for file synchronization.

Throughout the remainder of this section, we will focus on optimized techniques for the map construction phase, since good delta compression tools for the second phase are already available from several sources. Note that in the above example, hashes are transmitted from server to client, while in *rsync* the client sends hashes to the server. We could in principle also send hashes from client to server, and then have the server build a map of the client file f_{old}; at the end we could delta encode the entire file f_{new} with respect to the known parts of f_{old}. In fact, this is essentially what *rsync* does, in a single communication round. However, we would expect more bandwidth savings by sending hashes to the client and finally encoding only the unknown parts of f_{new}.

IEEE COMPUTER SOCIETY

5.2 Techniques for Efficient Map Construction

We now focus on techniques for efficient map contruction. While the example in the previous section might indicate that there is not much to do apart from repeatedly sending hashes, the problem turns out to be surprisingly rich in terms of possibilities. All the following techniques are based on exchanging hash values, but use various ideas to minimize the number of bits needed for the hashes. In particular we employ the following ideas, described in more detail further below:

(a) **Recursive splitting** of the block size by powers of 2. This technique was already used in [10, 25, 34]. We start out with a block size of 4096 or some other power of 2, and split any blocks that remain unmatched by a factor of 2 in each round. This technique is straightforward and does not require additional explanation.

(b) **Optimized match verification** to minimize the number of bits that are needed to verify, beyond a reasonable doubt, that substrings in the two files match. The idea is to first send a fairly weak hash that can be used to identify a possible match, and then use an optimized protocol based on ideas from group testing to filter out any false matches. A more limited version of this approach was also proposed in [25].

(c) **Local and continuation hashes** to decrease the number of bits that have to be initially sent in certain cases. In particular, continuation hashes are very weak hashes that are used to entend known matches towards the left and right in both files, allowing us to recurse down to block sizes for which "global" hashes (i.e, hashes that are matched against all positions in f_{old}) are too expensive. Local hashes trade off these two cases. The case of continuation hashes can be modeled as a version of the problem of "searching with liars", also called Ulam's problem [37, 49].

(d) **Decomposable hash functions** to decrease the number of bits used for the initial hashes for finding possible matches. The simple idea is that if we have already transmitted a hash value for the parent block and the left sibling, then for certain types of hash functions we could compute the hash value of the right sibling from the other two hashes. In practice though, designing appropriate hash functions to implement this is nontrivial.

In the remainder of this section, we describe these ideas in more detail. Subsection 5.6 summarizes the protocol.

5.3 Optimized Match Verification

Suppose that S initially sends a set of k-bit hashes to C, and that C uses these hashes to check for matches in f_{old}. Suppose that the block size is b and f_{old} is of length n. There are $n-b+1$ possible match positions in f_{old} that need to be checked, and for each position there is a certain probability of a false match. Thus, for a k-bit perfectly random hash value, the chance of a false match in f_{old} is

$$1-\left(1-\frac{1}{2^k}\right)^{n-b+1} \approx 1-e^{-\frac{n-b+1}{2^k}},$$

where e is the base of the natural logarithm. In short, around $\lg(n)$ bits per hash are needed to make sure that we have an even chance of not getting a false match, and any additional bit beyond $\lg(n)$ decreases the chance of a false match by roughly a factor of 2.

For web pages, $\lg(n)$ is already around 15 on average, and to be reasonably confident about a match we would need to send maybe another 32 bits (about the number *rsync* uses). Instead, we propose the following alternative: the server initially sends only slightly more than $\lg(n)$ bits to allow the client to identify candidates for matches; these candidates are then verified in an optimized manner, by having the client send verification hashes of its matches back to the server. We use three different ideas for this:

(i) The client can send one verification hash for each candidate back to the server. This hash is then compared only against the block that generated the original hash. The advantage is that verification hashes are only issued by the client for those hashes that found a possible match in f_{old}, which is typically a minority of hashes.

(ii) The client can send one verification hash for several candidate matches, in essence asking: *Are all these matches correct?* This should only be done once some degree of confidence is achieved, since one bad apple may cause the entire group to fail the test.

(iii) After a group match has failed, we can try to salvage some of the elements in the group by reissuing hashes for individual candidates or smaller groups.

Thus, we propose using a sequence of tests on various subsets of the candidates to efficiently identify those that are correct matches with fairly high certainty. This can be modeled by the following group testing problem [12], where false matches correspond to defective items and our goal is to identify all nondefective items, or a large subset of them, through a sequence of question of the following type: *Are all elements in a specified group nondefective?*

The answer to the question is unreliable in the following sense: If all elements are nondefective, the answer is always correct, but if there exists a defective item, then the correct answer is only returned with probability $1-\frac{1}{2^k}$. The cost of each question is $k+1$ (a k-bit hash for the subset and one bit for the reply), and the goal is to reliably identify nondefective items with minimum cost. We are not aware of previous results on this precise version of the problem. In practice, it appears that using only two or three batches of tests already gives close to optimal results, as we see later.

5.4 Local and Continuation Hashes

In the previous subsection, we saw that at least $\lg(n)$ bits are needed for a "global" hash that is compared to all po-

IEEE COMPUTER SOCIETY

sitions in the client file. However, in certain situations we can do better. Consider a scenario where we have already found a confirmed match between blocks $f_{new}[x, x+b-1]$ and $f_{old}[y, y+b-1]$. There is no reason to believe that the match stops exactly at the power-of-two block boundaries used by the algorithm, and thus we would like to try to extend the match towards the left and right in subsequent iterations with smaller block sizes. Thus, for block size $b' = b/2$, we could send a hash for blocks $f_{new}[x-b', x-1]$ and $f_{new}[x+b, x+b+b'-1]$ to the client to check for matches only with blocks $f_{old}[y-b', y-1]$ and $f_{old}[y+b, y+b+b'-1]$, respectively. In this case, even a very small number of bits (say, 3 or 4 per hash) would separate most of the true and false matches, and we could directly continue with the optimized match verification protocol described above. Since this makes the hashes much cheaper, we can afford to work with smaller block sizes than for global hashes.

This idea can be generalized to local hashes that use a little bit more than k bits and that are compared only with matches in a neighborhood of size 2^k in f_{old}. The neighborhood can be chosen with various heuristics based on confirmed matches of surrounding blocks of f_{new}, using the observation that most changes to files are fairly local in nature.

The problem of expanding matches, say, towards the right, can be modeled as a searching game with liars [37, 49] as follows: imagine performing a binary search with unreliable comparisons (corresponding to continuation tests with block sizes $b/2$, $b/4$ or $3b/4$, etc.) in the following sense: if the correct answer is ">" ("go to the right") then this answer is always returned; otherwise with probability $\frac{1}{2^k}$ a wrong answer is returned. The cost of each query is again $k+1$. Known results show that it is not optimal to verify each answer with a high probability before going down one level in the search tree. However, in our case we are performing many such searches concurrently, and we could perform group testing across these searches to more efficiently verify answers on each level; it is not clear which strategy is best in this case.

The same reasoning that motivated continuation matches also leads to another possible optimization. Suppose that a block at a particular level results in a confirmed match in the client file. In that case, it is unlikely that the sibling of this block would also find a match because (1) any match that is a continuation of the first match would have already been discovered at the parent level, and (2) any other match is unlikely since the match found by the first sibling will likely extend at least slightly into the other block. Thus, if we split the processing for each block size into two phases, first a search for matches using continuation hashes on blocks adjacent to confirmed matches, and then a search using global or local hashes, then in the second phase we can omit sending hashes for any blocks whose sibling found a confirmed match in the first phase (and also for any blocks for which continuation hashes were sent but no matches found). We could extend this idea by also first sending hashes, say, for all left siblings, and then only for those right siblings whose left sibling did not find a match. We did not implement this last idea since a technique described in the next subsection already avoids sending hashes for both siblings in this case. We did implement the idea of first sending continuation hashes, and then global hashes, and observed some moderate benefits.

5.5 Decomposable Hash Functions

We say that a hash function h is composable if we can efficiently compute $h(f[l,r])$ from the values $h(f[l,m])$, $h(f[m+1,r])$, $m-l$, and $r-m-1$. A hash function is decomposable if we can efficiently compute $h(f[m+1,r])$ from the values $h(f[l,r])$, $h(f[l,m])$, $r-l$, and $r-m-1$, and also $h(f[l,m])$ from $h(f[l,r])$, $h(f[m+1,r])$, $r-l$, and $m-l$. A decomposable hash function allows us to significantly save on the cost of the hashes for identifying candidate for matches: since we have already transmitted a hash for the parent block, we only have to send one additional hash per pair of sibling blocks – the hash for the other sibling can then be computed from these two.

In practice, there are some additional obstacles. Since our other techniques use different numbers of bits for different blocks, we ideally would like to have a hash function that is "bit-prefix" decomposable in the sense that from the first s bits of $h(f[l,r])$ and $h(f[l,m])$ we would like to be able to compute the first s bits of $h(f[m+1,r])$, for any s. In addition, there are several other desirable properties of a hash function. It should be "rolling" so that $h(f[l+1,r+1])$ can be computed in constant time from $h(f[l,r])$, and it should be strong in the sense that different strings should have a probability close to $\frac{1}{2^k}$ of mapping to the same hash value, for a k-bit hash. Finally, strings that can be obtained from each other through permutation should not be mapped to the same hash too often, as such cases are quite common in practice. Some of the techniques used in practice are also limited to certain ranges of block sizes. There are trade-offs between some of these properties, and in the end we designed our own modification of the Adler checksum in *rsync*, which appears to perform well on all our data sets and block sizes.

5.6 Overview of the Protocol

We integrate all of the above techniques into a protocol consisting of a sequence of rounds, one for each block size. Each round consists of one or more roundtrips of communication. Consider the left part of Figure 5.2 for an example. The round starts with the server sending a set of hashes to the client; these hashes may be global, local, or continuation hashes and are usually strong enough to identify candidates for matches, but not strong enough to reliably verify the matches. The decomposability of the hash function is implemented at a lower level by surpressing the transmission of hash bits that can be computed from sibling and ancestor hashes. The client then replies with a bitmap specifying which hashes found a match candidate, immediately followed by a set of verification hashes for the candidates. Each verification hash, based on MD5, can be for a single candidate or a group of candidates. The server than receives and checks the verification

IEEE COMPUTER SOCIETY

hashes, and replies with a bitmap specifying which verification hash was confirmed. If the round consists of a single batch of verification hashes, then this bitmap is included into the first roundtrip of the next round, and is immediately followed by hashes for the next smaller block size. Otherwise, the client may send additional batches of verification hashes.

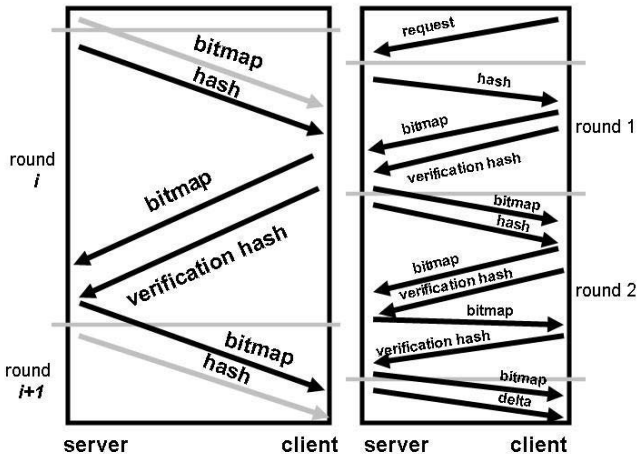

Figure 5.2. A single round of the protocol (left), and a complete protocol (right) consisting of the client request, one round with one batch of verification hashes, one round with two batches of verification hashes, and the final delta sent from server to client.

On the right side of Figure 5.2 we see a simple example consisting of two rounds. The client first sends a request to the server, who then sends hashes for the first block size. In the example, the first round has a single batch of verification hashes, while the second round involves two batches, maybe first a set of weak hashes for each candidate, and then stronger hashes for small subsets of 4 or 8 surviving candidates. Finally, the server sends a delta to the client. In our implementation, a simple parameter file is used to specify all the options and techniques that should be used in each round, such as the type and number of bits per hash, the strategy for verifying candidate hashes through individual or group hashes or for salvaging failed candidates, etc.

6 Experimental Evaluation

We now report on a preliminary experimental evaluation of our prototype software. We first describe the experimental setup in terms of implementation, data sets, and the other tools that we compare against. Subsection 6.2 provides results on two data sets previously used to evaluate delta compression tools, and Subsection 6.3 looks at the case of our web page update application. We note that our work is still ongoing and thus some numbers may improve slightly in subsequent versions.

6.1 Experimental Setup

We implemented a prototype with the techniques from the previous section in around 1400 lines of `C` code. The program was compiled with `gcc` and was run on several Linux servers. The prototype was not optimized in terms of CPU performance yet, and thus we do not focus too much on this aspect. We used the *zdelta* delta compressor[2] for the delta compression step, and used a very simple but efficient hash table based on double hashing to search for matching blocks. The prototype can be configured easily to vary the number of rounds and the set of techniques that are applied in each round, allowing us to look at the impact of each technique.

For the hashes sent from server to client, we used two implementations: a high quality one based on MD5 that is not rolling or decomposable (thus requiring significantly more running time and moderately more bandwidth) and a more heuristic construction of a hash function that is both rolling and decomposable. For the verification hashes, we used another MD5-based hash in both implementations, since in this case we do not need the other properties. As in *rsync*, there is a small probability of failure on each file. Such failures are detected through the exchange of a very strong 16-byte hash value for each file in the beginning; this also allows our code to detect unchanged files at that point.

We compare our prototype implementation to *rsync* with default block size, *rsync* with an optimally chosen block size for each individual file, and the *zdelta* [46] and *vcdiff* [23] delta compressors. We note that current delta compressors are already fairly optimized in terms of compression, and thus provide a reasonable bound on what we could hope to expect from a file synchronization tool that does not have access to the outdated file (guarding a major breakthrough in delta compression techniques). We used the following data sets for our evaluation:

1. The *gcc* and *emacs* data sets used for evaluating delta compression performance in [21, 45], 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.

2. A collection of ten thousand web pages that we crawled repeatedly every night for several weeks during Fall 2001. Pages were selected at random from two massive web crawls of hundreds of millions of pages and are thus a fairly reasonable random sample of the web. We use four versions of each page: a base set and three updated versions crawled 2, 20, and 72 days later, respectively. In the experiments, we measure the cost of updating the base set to one of the updated sets.

[2] Available at `http://cis.poly.edu/zdelta/`

IEEE COMPUTER SOCIETY

6.2 Performance on Benchmark Data Sets

We first look at the performance of a very basic version of our protocol that uses a decomposable hash function, recursive halving of blocks, and a separate verification hash for each candidate match, but none of the other techniques. In Figures 6.1 and 6.2, we give results for the *gcc* and *emacs* data sets, respectively. For each block, our protocol sends a hash of $\lg(n) + 4$ bits to identify candidates, and the client replies with a 32-bit verification hash for each match that is found. The initial block size of our protocol is 32768 bytes, and we plot the total cost in KB for the entire data set under different minimum block sizes after which we terminate the recursion. We also show results for *rsync* with default block size, an idealized *rsync* that knows the best block sizes for each file, and the *zdelta* delta compressor.

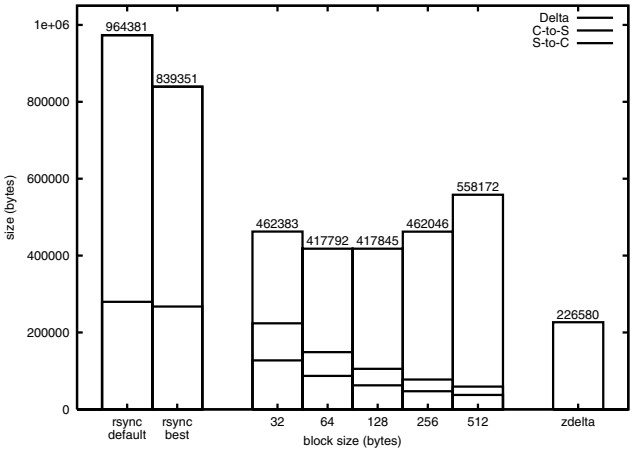

Figure 6.1. Performance of the basic protocol with different minimum block sizes on the *gcc* data set, compared to *rsync* and *zdelta*. For *rsync*, we show the costs of client-to-server and server-to-client communication, and for our protocol we show the cost of server-to-client and client-to-server communication during the map construction phase and the cost of the final delta (in order from bottom to top).

As we see, recursive halving and the use of decomposable hashes and verification hashes already gives significant benefits. The recursion should be stopped around a block size of 128 or 64 bytes for best results. (At that point, the increase in the cost of the map construction phase becomes higher than the decrease in the delta compression phase.) However, the best result is still about a factor of 2 away from the performance of the highly optimized delta compressor. We note that without decomposable hash functions, the amount of data sent from server to client in the map building phase would be about twice as high, and as a result the optimal minimum block size is also slightly larger in that case.

Next, we consider the benefit of using an improved match verification approach and continuation hashes in the protocol. In particular, we send 6-bit continuation hashes for all

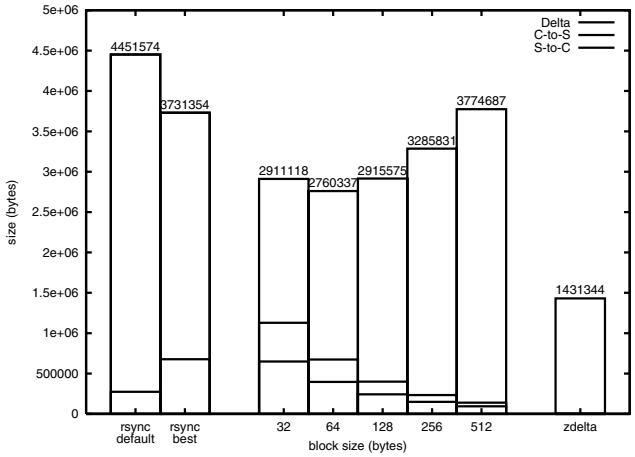

Figure 6.2. Performance of the basic protocol with different minimum block sizes on the *emacs* data set.

blocks adjacent to an already matched block and $\lg(n) + 6$ bits for all other blocks (in the same roundtrip). The client then sends one 32-bit verification hash for every group of 5 candidate matches. We chose two close-to-optimal minimum block sizes for the global hashes according to the previous experiment, and then continue with continuation hashes down to sizes of 64, 32, 16, and 8 bytes. The results are shown in Figure 6.3.

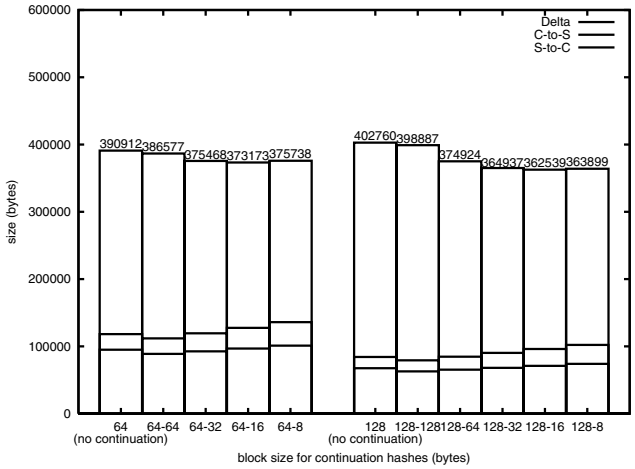

Figure 6.3. Performance of the protocol with continuation hashes of various minimum block size. Shown in the leftmost bar of each group is the performance with no continuation hashes but with group verification.

We see from the leftmost bars in the two groups that the simple single-roundtrip group verification already gives some improvements over the corresponding numbers in Figure 6.1. For the *gcc* data sets, the best choice appears to be to use continuation hashes down to 16-byte blocks; for *emacs* it would be 8-byte blocks (not shown). Also, we see that when using continuation hashes down to 16-byte blocks, having a mini-

IEEE COMPUTER SOCIETY

mum block size for global hashes of 128 bytes becomes better than 64 bytes. We also experimented with the use of local hashes as described in Subsection 5.4; however, we were unable to get any significant improvements from this technique. One issue here is the "harvest rate", i.e., the percentage of hashes that result in confirmed matches. Not surprisingly, blocks that qualify for continuation hashes have a fairly high harvest rate, which is another reason why they can be profitably used for much smaller block sizes. Local hashes do not fare well in this context, though we plan to revisit this issue in future experiments.

We now look at the impact of the optimized match verification techniques from Subsection 5.3. In particular, we compare the following five strategies (note that we always send $\lg(n)$ bits less from server to client for continuation hashes):

(1) the trivial verification with 32-bit hashes for each candidate, used in the first experiment,

(2) the slightly smarter approach above where we used $\lg(n) + 6$ bits for each global hash sent from server to client, and then return one 32-bit hash for every group of 5 candidates (losing all 5 candidates if the verification hash fails),

(3) an approach with one additional roundtrip per round where we first send $\lg(n) + 3$ bits for each global hash, then send a 6-bit verification hash for each candidate back, and then one 32-bit verification hash for each group of 16 surviving candidates,

(4) a more complicated approach with three roundtrips per round where we send only $\lg(n)+2$ bits per global hash to the client, who first replies with a 5-bit hash for each match, then with a 3-bit hash for groups of 3, and finally with a 32-bit hash for groups of 32, and

(5) an even more complicated approach with 4 roundtrips per round where we first send $\lg(n) + 2$ bits, and a 5-bit verification hash for each candidate back, then an 8-bit hash for groups of 4, then a salvage phase where we send 9 bits for each candidate in a failed group, and finally a 32-bit hash for groups of 32.

The results in Figure 6.4 show slight improvements for each method for *gcc* (the same holds for *emacs*). However, almost all benefits are obtained with only one or two roundtrips. We experimented with a number of other settings, but none did significantly better than the best one shown. In particular, we did not find any benefit in being very aggressive about grouping large numbers of fairly uncertain candidates and then trying to salvage candidates from failed groups. Instead, it appears to be preferable to slowly grow the size of the groups as our confidence in the candidates grows. There are a number of subtle tradeoffs at work here: for example, decreasing the number of bits sent to the server from $\lg(n) + 2$ to $\lg(n) + 1$ and increasing the reply by one bit does save bits at this point,

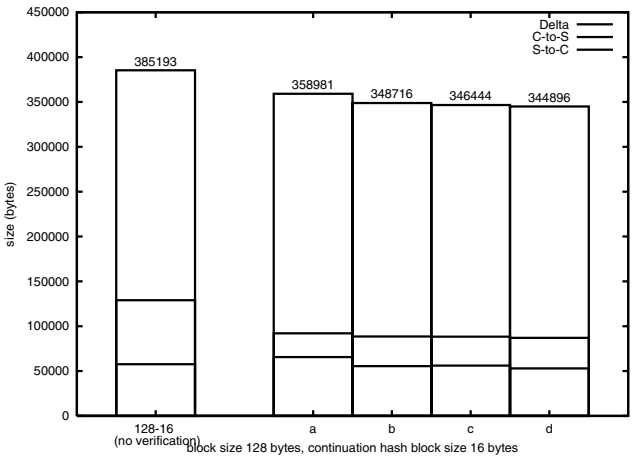

Figure 6.4. Performance of different techniques for match verification on the *gcc* data set. Trivial verification is shown in the leftmost bar, while the other bars show optimized solutions with 1, 2, 3, and 4 roundtrips per round as described. The minimum block size is 128 bytes for global hashes and 16 bytes for continuation hashes.

	gcc	emacs
number of files	1,002	1,286
uncompressed	27,288	27,326
gzip	7,563	8,577
rsync default	964	4,452
rsync best	839	3,731
zdelta	227	1,431
our results	336	2,343

Table 6.1. Best results for the *gcc* and *emacs* data sets using all techniques (in KB).

but results in some real matches being lost due to false positives taking their place, and ultimately a larger delta.

There are several other minor optimizations that we implemented, such as first sending continuation hashes and then global hashes in the next roundtrip, or the selective use of local hashes. None of these showed significant improvements. In Table 6.1 we show the best results that we were able to obtain by using all techniques; we note that this results in a total of more than 50 roundtrips and is thus probably not good in practice since each roundtrip also requires some computation (and sometimes a scan of the files).

Looking at Table 6.1, we see that we achieve bandwidth savings of a factor of 1.9 to 2.8 over *rsync* with our new techniques, and that we are within 45% to 70% of the best compression achieved by the delta compressor. A note concerning computation times: we have not yet optimized our code in terms of CPU performance but expect significant fu-

IEEE COMPUTER SOCIETY

	2 days	20 days	72 days
uncompressed size	143 MB	143 MB	140 MB
number of files	10,000	10,000	10,000
number of changed files	2,818	3,747	5,127
rsync (default)	5,339	7,766	11,770
zdelta	1,479	2,170	3,879
our results	2,062	3,623	6,703

Table 6.2. Cost of updating a web collection using various methods, for various update frequencies. The cost is in KB for 10000 web pages.

ture improvements. The purpose of the work presented here is to show the potential benefits of the various techniques, and we plan to build a high performance version based on our conclusions. The prototype currently runs at a speed of up to a few MB of raw data per second; with a compression ratio of $80:1$ as for *gcc* this results in a data rate of only a few hundred kbits/s over the network. For faster networks and highly redundant data sets, CPU performance would currently be a bottleneck. Finally, our current implementation is also not optimized in terms of memory consumption.

6.3 Performance in a Web Page Update Application

We now look at the web page update application that motivated our work. Recall that we are given ten thousand pages, selected at random from the web, that were recrawled every night for several weeks. Each set has a total size of around 140 MB, with about 14 KB per page on average. Some of the files are not updated at all between crawls, while others change only slightly. In Table 6.2, we show the bandwidth cost of maintaining such collections at a client using our techniques, under different update frequencies.

We observe that our techniques support the maintenance of very large replicated sets of web pages even over fairly slow links, and improve over *rsync* by nearly a factor of 2. For example, if we synchronize the pages every two days then slightly more than 2 MB of data transfer suffices to maintain 10000 pages at a client PC, which is easily done over cable or DSL links. The result shown are the best we could get for our protocol with all optimizations, but as before there are simpler settings with fewer roundtrips that perform within 3 to 5% of optimal.

7 Concluding Remarks

There are a number of unresolved issues and open problems left by our work. First, the current implementation is a very early prototype, and we have not optimized it in terms of CPU performance. This may result in the CPU becoming a bottleneck for faster networks. While some overhead is due to the repeated passes over the data in the different communication phases, we believe that significant improvements are possible through careful optimization. Some minor additional improvements in bandwidth efficiency should also be possible.

We plan to integrate our implementation into a system for maintaining large sets of changing web pages over wide area networks. We also intend to use the presented techniques as the basis for a new general purpose tool for file synchronization over slow links that we plan to release. Ideally, such a tool would be adaptive and thus choose the best set of parameters and number of roundtrips based on the characteristics of the data set and communication link.

On the theoretical side, we are working on improved asymptotic bounds for file synchronization under some common file similarity metrics. Interestingly, the idea of continuation hashes used in this paper appears to be very promising in this context as well. A detailed study of the group testing and "searching with liars" problems discussed in this paper might also lead to slight improvements in practice.

We are also studying how to improve file synchronization if we are restricted to just one or two round-trips. In this case, it seems difficult to improve significantly over *rsync* in practice, but we believe that at least some moderate gains are possible. Finally, we plan to look at synchronization in asymmetric cases, e.g., in cases with server broadcast capability, lower upload speed, or a bottleneck at a busy server.

References

[1] K. Abdel-Ghaffar and A. E. Abbadi. An optimal strategy for comparing file copies. *IEEE Transactions on Parallel and Distributed Systems*, 5(1):87–93, Jan. 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, Oct. 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, Apr. 1991.

[5] O. Brandman, J. Cho, H. Garcia-Molina, and N. Shivakumar. Crawler-friendly web servers. In *Proc. of the Workshop on Performance and Architecture of Web Servers (PAWS)*, June 2000.

[6] B. Brewington and G. Cybenko. Keeping up with the changing web. *IEEE Computer*, 33(5), May 2000.

[7] J. Cho and H. Garcia-Molina. The evolution of the web and implications for an incremental crawler. In *Proc. of 26th Int. Conf. on Very Large Data Bases*, pages 117–128, Sept. 2000.

[8] J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshness. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 117–128, May 2000.

[9] J. Cho and A. Ntoulas. Effective change detection using sampling. In *Proc. of the 28th Int. Conf. on Very Large Databases*, pages 514–525, Sept. 2002.

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

IEEE COMPUTER SOCIETY

[11] 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.

[12] R. Dorfman. The detection of defective members in large population. *Annals of Mathematical Statistics*, 14:436–440, 1943.

[13] F. Douglis, A. Feldmann, B. Krishnamurthy, and J. Mogul. Rate of change and other metrics: a live study of the World Wide Web. In *Proc. of the USENIX Symp. on Internet Technologies and Systems (ITS-97)*, pages 147–158, Berkeley, Dec. 8–11 1997. USENIX Association.

[14] J. Edwards, K. McCurley, and J. Tomlin. An adaptive model for optimizing performance of an incremental web crawler. In *Proc. of the 10th International World Wide Web Conference*, pages 106–113, May 2001.

[15] 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, Jan. 1998.

[16] J. Fiedler and J. Hammer. Using the web efficiently: Mobile crawlers. In *Proceedings of thw 17th Int. Conf. on Computer Science*, San Diego, CA, 1999.

[17] H. Garcia-Molina. Webbase: Building a web warehouse. In *Plenary Talk at the 2003 Federated Computing Research Conference*, June 2003.

[18] J. Gray, W. Chong, T. Barclay, A. Szalay, and J. Vandenberg. Terascale sneakernet: Using inexpensive disks for backup, archiving, and data exchange. Technical Report MSR-TR-2002-54, Microsoft Research, May 2002.

[19] Grub. Distributed Internet Crawler. http://www.grub.org/.

[20] J. Hirai, S. Raghavan, H. Garcia-Molina, and A. Paepcke. WebBase : A repository of web pages. In *Proc. of the 9th Int. World Wide Web Conference*, May 2000.

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

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

[23] 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.

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

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

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

[27] 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.

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

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

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

[31] 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.

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

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

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

[35] 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.

[36] 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.

[37] A. Pelc. Searching games with errors - fifty years of coping with liars. *Theoretical Computer Science*, 270(1-2):71–109, Jan. 2002.

[38] C. Pyne. Remote file transfer method and apparatus, 1995. US Patent Number 5446888.

[39] 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.

[40] D. Rasch and R. Burns. In-place rsync: File synchronization for mobile and wireless devices. In *Proc. of the USENIX Annual Technical Conference*, June 2003.

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

[42] 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.

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

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

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

[46] 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.

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

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

[49] S. Ulam. *Adventures of a Mathematician*. Scribner, 1976.

IEEE COMPUTER SOCIETY