Optimizing File Replication over Limited-Bandwidth Networks using Remote Differential Compression

Dan Teodosiu, Nikolaj Bjørner, Yuri Gurevich, Mark Manasse, Joe Porkka

{danteo, nbjorner, gurevich, manasse, jporkka} @ microsoft.com
Microsoft Corporation

Abstract— **Remote Differential Compression (RDC) protocols can efficiently update files over a limited-bandwidth network when two sites have roughly similar files; no site needs to know the content of another's files a priori. We present a heuristic approach to identify and transfer the file differences that is based on finding similar files, subdividing the files into chunks, and comparing chunk signatures. Our work significantly improves upon previous protocols such as LBFS and RSYNC in three ways. Firstly, we present a novel algorithm to efficiently find the client files that are the most similar to a given server file. Our algorithm requires 96 bits of meta-data per file, independent of file size, and thus allows us to keep the metadata in memory and eliminate the need for expensive disk seeks. Secondly, we show that RDC can be applied recursively to signatures to reduce the transfer cost for large files. Thirdly, we describe new ways to subdivide files into chunks that identify file differences more accurately. We have implemented our approach in DFSR, a state-based multi-master file replication service shipping as part of Windows Server 2003 R2. Our experimental results show that similarity detection produces results comparable to LBFS while incurring a much smaller overhead for maintaining the metadata. Recursive signature transfer further increases replication efficiency by up to several orders of magnitude.**

I. INTRODUCTION

As the amount of data shared over the Internet continues to grow rapidly, users still experience high costs and long delays in transferring large amounts of information across the network.

However, it often happens that a large fraction of the information that is transmitted is redundant, as the recipient may already have stored a similar (if not identical) copy of the data. 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 differences 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.

Delta-compression utilities such as diff [1][11][13], vcdiff [17], xdelta [21], BSDiff [25], or zdelta [32] may be used to produce a succinct description of the differences between two files if both files (the old and the new version) are locally available to a sender. However, in many distributed systems this assumption may be overly restrictive, since it is difficult or infeasible to know which old copies of the files (if any) other nodes hold. (A notable exception is the case of software distribution, where the sender may store the previous versions of the binaries and pre-compute differences).

A different class of protocols can be used if the old and the new version of the file are at the opposite ends of a slow network connection. These *Remote Differential Compression* (RDC) protocols heuristically negotiate a set of differences between a recipient and a sender that have two sufficiently similar versions of the same file. While not as precise as local delta compression, RDC may help to greatly reduce the total amount of data transferred.

In the Low Bandwidth File System (LBFS) [24], an RDC protocol is used to optimize the communication between a sender and a recipient by having both sides subdivide all of their files into chunks and compute strong checksums, or signatures, for each chunk. When a client needs to access or copy a file from the server, the latter first transmits the list of signatures for that file to the client, which determines which of its old chunks may be used to reconstruct the new file, and requests the missing chunks. The key to this protocol is that the files are divided independently on the client and server, by determining chunk boundaries from data features. Compared to chunking at fixed boundaries (an approach used by RSYNC [25][34]), this data-dependent chunking opens up other applications, such as using a system-wide database of chunk signatures on the client.

This paper builds upon the LBFS approach in the context of DFSR, a scalable state-based multi-master file synchronization service that is part of Windows Server 2003 R2. Some of the primary uses of DFSR include the distribution of content from a small number of hubs to a large number of spoke nodes, the collection of content from spokes back to the hubs for backup and archival purposes, and ad-hoc collaboration between spokes. Hubs and spokes may be arranged in a user-defined and dynamically modifiable topology, ranging up to a few thousand nodes. In most actual configurations, spokes will be geographically distributed and will often have a limited-bandwidth connection to the rest of the system; satellite links or even modem-based connections are not uncommon. Therefore, efficient use of connection bandwidth is one of the foremost customer requirements for DFSR.

In our RDC implementation, we significantly improve upon LBFS as well as other similar protocols, such as the widely used RSYNC protocol [25][34], in three different ways.

The first contribution is a novel and very efficient way for allowing a client to locate a set of files that are likely to be *similar* to the file that needs to be transferred from a server. Once this set of similar files has been found, the client may reuse any chunks from these files during the RDC protocol. Note that in the context of multi-master replication, we use the terms “client”, instead of recipient, and “server”, instead of sender, to indicate the direction of file synchronization, thus a given node may act as both a client and a server for different transactions, even at the same time.

Our similarity approach differs from LBFS in a significant way. Whereas LBFS maintains a database of chunk signatures across all the files on the node, RDC maintains only a small fixed size (96 bits) piece of meta-data per file. Since we may exclude meta data for very small files (in DFSR, this is set to less than 64KB), an index of this data can easily fit in main memory, even on large shares with more than 1M such files. LBFS on the other hand has to perform a tradeoff between the average chunk sizes and the size of the data-base. An average chunk size of 8KB, as suggested in [24], results in only 0.4% overhead for the chunk database, assuming 32 bytes for signatures and offset per chunk. However, chunk sizes in the order of 200 bytes, which in our experiments were found to be much more effective for differential compression would require an unacceptable 15% overhead for the database alone. The combination with recursion, furthermore amplifies the benefits of our similarity based approach.

The second contribution is that the LBFS RDC protocol can be applied *recursively*, by treating the signatures generated as a result of chunking as a new input to the protocol. This results in a second level of signatures that are transmitted in order to reconcile the first level of signatures. The first level of signatures is finally used to reconcile the real file contents. Recursion can of course be applied to an arbitrary depth. The benefit of recursion is that it reduces the signature transfer cost of the protocol, which can be significant for large files.

The third contribution is a chunking algorithm that identifies file differences more accurately, and an analysis of its quality. LBFS summarizes small windows of a file using a hash; chunk boundaries are chosen when the hash is divisible by a fixed base m. The average chunk size is thus equal to m when the hashes are evenly distributed. Neither the minimal nor the maximal chunk size is determined by this method. It is possible to impose a minimal chunk size by disregarding all hashes within a given minimal distance from a previous chunk boundary but this negatively affects the number of chunks that can be expected to coincide for roughly similar files.

Our *local maxima* chunking algorithm is also based on examining an interval of hash values. A position is a cut point, or chunk boundary, if its hash is larger than the hashes at all other surrounding positions within distance h. We show that determining local maxima is asymptotically no more expensive than determining cut-points by the other methods. In particular, we give an algorithm that requires only $1+\ln(h)/h$ comparisons per position. The local maxima chunking approach has the advantage of a “built-in” minimal chunk length, and of not requiring auxiliary parameters. A probabilistic analysis reveals that this approach also recovers more quickly from file differences.

Based on a large corpus of files and using the production version of DFSR, we show that the bandwidth savings achieved using our similarity detection approach are within just a few percent of LBFS, while incurring a tiny fraction of the cost for maintaining the metadata.

Further experimental results over a large collection of files show that RDC with recursion is significantly more efficient than RSYNC in most cases. For some very large files RDC uses up to four *times* less bandwidth than RSYNC. Although we expected it to be not as efficient, RDC compares favorably with local differential algorithms.

The rest of this paper is structured as follows. Section II summarizes the basic LBFS RDC protocol [24]. Section III presents our file similarity detection technique. Section IV describes the recursive signature transfer. Section V discusses our improved chunking algorithm. Section VI details the implementation of these techniques in DFSR. Section VII presents experimental results. The paper concludes after a discussion of related work.

II. The Basic RDC Protocol

For completeness, we summarize in this section the basic RDC protocol used in LBFS [24]. 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 F_C, as this shall facilitate the presentation of our approach in the following sections. Readers familiar with the LBFS algorithm may skip to the end of this section.

Referring to Fig. 1, the basic RDC protocol assumes that the file F_S on the server machine S needs to be transferred to

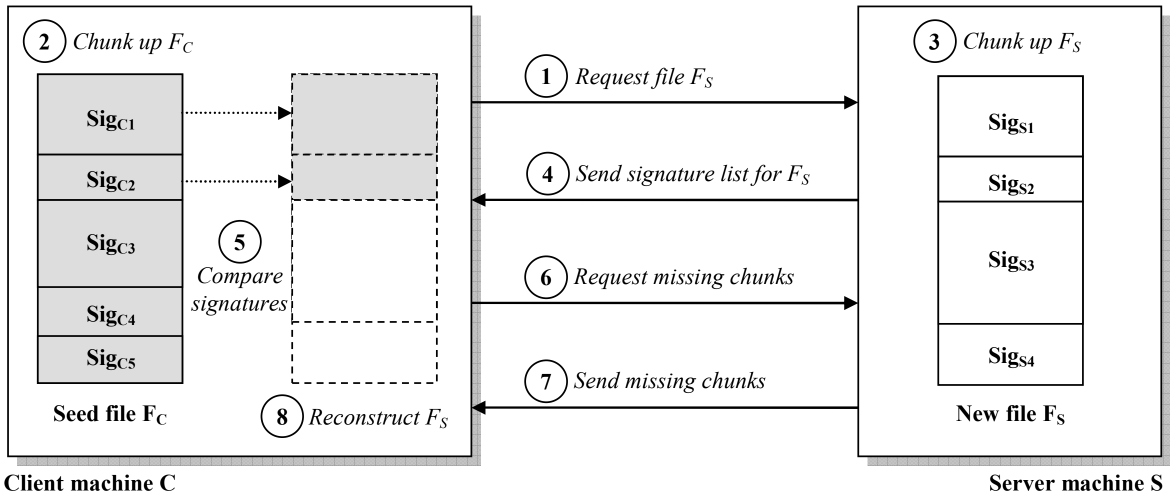

Fig. 1. The basic RDC protocol in LBFS [24] starts with both client and server machines dividing their respective files into chunks and computing signatures for each chunk. The server then communicates the list of signatures to the client, which checks it against its own to determine which chunks it needs to request from the server. Upon receipt of the missing chunks, the client combines the received chunks with its own seed ones to reassemble a copy of the new file F_S.

the client machine C using the seed file F_C stored on the client. F_S is a new version containing incremental edits over the seed file F_C. The transfer of F_S from S to C is performed as follows:

Step 1: C sends S a request to transfer file F_S.

Step 2: C partitions F_C into *chunks* by using a fingerprinting function that is computed at every byte position of F_C. A *chunk boundary* is determined in a data-dependent fashion at positions for which the fingerprinting function satisfies a certain condition. Next, a signature Sig_{Ck} is computed for each chunk *k* of F_C. 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 F_S into chunks and computes the chunk signatures Sig_{Sj}. Steps 2 and 3 may run in parallel.

Step 4: S sends the ordered list of chunk signatures and lengths $((Sig_{S1},Len_{S1}) \ldots (Sig_{Sn},Len_{Sn}))$ to C. Note that this implicitly encodes the offsets of the chunks in F_S.

Step 5: As this information is received, C compares the received signatures against its own set of signatures $\{Sig_{C1},\ldots,Sig_{Cm}\}$ computed in Step 2. C records every distinct signature value received that does not match one of its own signatures Sig_{Ck}.

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 F_S.

Step 7: S sends the content of the requested chunks to C.

Step 8: C reconstructs F_S by using the chunks received in Step 7, as well as its own chunks of F_C that in Step 5 matched signatures sent by S.

In LBFS, the entire client file system acts as the seed file F_C. 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 following three sections describe this similarity approach and two additional enhancements that we have made to the RDC algorithm.

III. Using Similarity Detection to Find RDC Candidates

A. *Finding RDC candidates*

To transfer a file F_S from the server to the client, the RDC protocol requires that a seed F_C that is similar to F_S be identified on the client. In some cases, a simple heuristic based on file identity, such as equality of the file path or of the file unique identifier assigned by a state-based replicator, can be used to identify F_C. This assumes that the new version F_S was derived from the old version F_C through incremental edits.

However, in many cases a seed file cannot be identified *a priori*, although the client may already store several good RDC candidates. For example, if a new file F_N is created on the server S by simply copying F_S, but no record of the copy operation is kept, then it may be difficult to determine that F_C

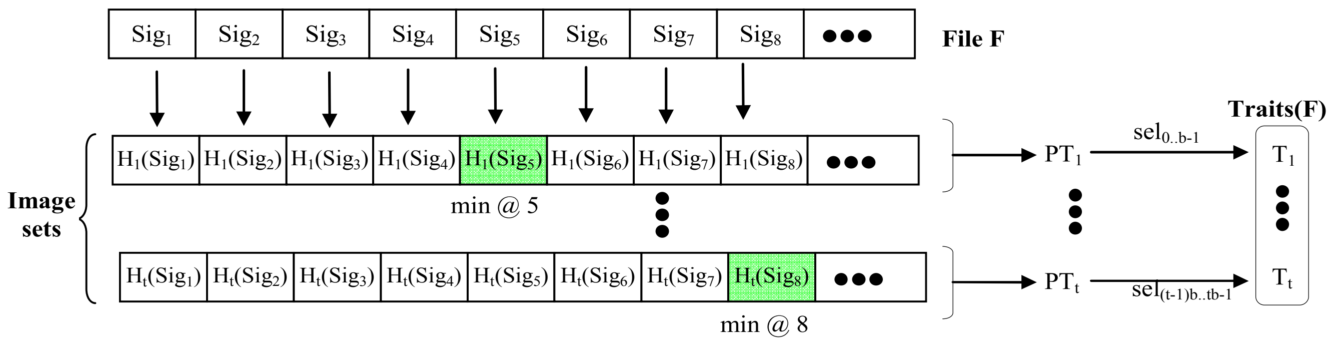
Sig1 Sig2 Sig3 Sig4 Sig5 Sig6 Sig7 Sig8
File F
Image sets
H1(Sig1) H1(Sig2) H1(Sig3) H1(Sig4) H1(Sig5) H1(Sig6) H1(Sig7) H1(Sig8)
min @ 5
Ht(Sig1) Ht(Sig2) Ht(Sig3) Ht(Sig4) Ht(Sig5) Ht(Sig6) Ht(Sig7) Ht(Sig8)
min @ 8
PT1
PTt
sel0..b-1
sel(t-1)b..tb-1
Traits(F)
T1
Tt

Fig. 2. Computing the traits of a file F is done by mapping the signatures into *t* image sets through *t* different hash functions. A pre-trait is derived out of each image set by taking the element with minimum hash value from the set. Traits are computed by selecting *b* bits out of each pre-trait.

is the perfect seed for the differential transfer of F_N. Examples of this scenario can be frequently observed in environments where DFSR is deployed; examples include saving a large modified PowerPoint presentation to a new file, making a copy of an existing large document, creating a new virtual machine image by installing an application on top of an existing image, etc.

In this section, we describe a technique that allows the client C to efficiently select a small subset of its files, F_{C1}, F_{C2}, …, F_{Cn}, that are similar to a file F_S that needs to be transferred from S using the RDC protocol. Typically, the set of similar files to be chosen will be small, i.e. $n \le 10$.

Similarity between two files F_S and F_C is measured in terms of the number of identical chunks that F_S and F_C share, divided by the total number of distinct chunks in the two files. If Chunks(F_S) and Chunks(F_C) are the sets of chunks computed for F_S and F_C in Steps 3 and 2 of the base RDC protocol, then:

$$\mathrm{Sim}(F_S, F_C) = \frac{\left|\mathrm{Chunks}(F_S) \cap \mathrm{Chunks}(F_C)\right|}{\left|\mathrm{Chunks}(F_S) \cup \mathrm{Chunks}(F_C)\right|}$$

As a proxy for chunk equality we shall use equality on the signatures of the chunks. If the signatures are computed using a cryptographically secure hash function (such as SHA-1 or MD4), this is highly accurate, due to the extremely low probability of a hash collision. Thus, if Signatures(F_S) and Signatures(F_C) are the sets of signatures for F_S and F_C, then:

$$\mathrm{Sim}(F_S, F_C) \cong \frac{\left|\mathrm{Signatures}(F_S) \cap \mathrm{Signatures}(F_C)\right|}{\left|\mathrm{Signatures}(F_S) \cup \mathrm{Signatures}(F_C)\right|}$$

One fine point to note in the above is the fact that a file may contain several identical chunks (with identical signatures). This is irrelevant from the point of view of the RDC protocol, as identical chunks will be requested and transferred at most once in Steps 6 and 7, respectively.

Given a file F_S and Files_C, the set of all the files that are stored on C, the problem we need to solve is to identify the files in Files_C that have the highest degree of similarity with F_S. If F_{C1}, F_{C2}, ..., F_{Cn} are the *n* files in Files_C most similar to F_S, we define:

$$\mathrm{Similar}(F_S, \mathrm{Files}_C, s) = \{F_{C1}, F_{C2}, ..., F_{Cn}\}$$

where by “most similar” we mean that for all files F_{Ci}, Sim(F_S, F_{Ci}) $\ge s$, and for all other files x in Files_C, Sim(F_S, x) $\le$ Sim(F_S, F_{Ci}). *s* is a similarity threshold explained below.

A brute-force approach for computing Similar(F_S, Files_C, *s*) could be based on the set of signatures sent by S in Step 4 of the RDC protocol. However, there are two problems with this approach. Firstly, C would need to wait until it received at least a substantial portion of the signature list before selecting the similar files and starting to execute Step 5, thus significantly reducing its pipelining opportunities. Secondly, lacking a seed, the recursive signature transfer described in Section IV could not be applied to the signature list.

B. *Using traits to encode similarity information*

File similarity can be approximated by using the following heuristic that makes use of a compact summary of a file’s signatures, called its set of *traits*. The set of traits Traits(F) of a file F can be computed simultaneously with the partitioning of the file into chunks during Step 2 of the RDC algorithm, and cached as part of the file metadata.

The basic RDC protocol described in the previous section is modified as follows to allow the client C to identify and use as a seed the set of *n* files F_{C1}, F_{C2}, …, F_{Cn} that are similar to the file F_S to be differentially transferred:

Step S.1.1: C sends S a request to transfer the file F_S.

Step S.1.2: S replies with the set of traits for F_S, Traits(F_S). Traits(F_S) will be usually cached as part of the metadata of F_S, and thus can be sent to C without additional overhead.

Step S.1.3: C uses Traits(F_S) to identify a set of its existing files, F_{C1}, F_{C2}, …, F_{Cn}, that are likely to be similar to F_S.

Step S.2: C partitions all identified files F_{C1}, F_{C2}, …, F_{Cn} into chunks and computes a signature Sig_{Cik} for each chunk k of each file F_{Ci}.

Step S.3: S partitions F_S into chunks and computes the chunk signatures Sig_{Sj}.

Step S.4: S sends the ordered list of chunk signatures and lengths ((Sig_{S1},Len_{S1}) … (Sig_{Sn},Len_{Sn})) to C.

Step S.5: As this list is received, C compares the received signatures against its own set of computed signatures (Sig_{C11},…Sig_{C1m},…Sig_{Cn1},…). C records every distinct signature value received that does not match one of its own signatures Sig_{Cik}.

Step S.6: C sends a request to S for all the chunks for which there was no matching signature on C.

Step S.7: S sends the requested chunks to C.

Step S.8: C reconstructs F_S by using the chunks received in Step S.7, as well as its own chunks of F_{C1}, F_{C2}, …, F_{Cn} that in Step S.5 matched signatures sent by S. Once a copy of F_S has been reconstructed, C adds F_S to its collection of stored files and stores Traits(F_S) as part of its metadata.

To minimize network traffic and CPU overhead, it is essential that Traits(F_S) be very small and that the set of similar files F_{C1}, F_{C2}, …, F_{Cn} be determined quickly.

The algorithm for identifying similar files has two main parameters (b,t) that are summarized below. These parameters are explained in detail in the following description.

b	:	Number of bits per trait
t	:	Number of traits per object

C. Computing the set of traits for a file

Fig. 2 shows an example of a trait computation. Traits(F) is derived from the chunk signatures of F as follows:

Step T.1: The signature set {Sig_1…Sig_n} is mapped into t image sets by applying t different hash functions H_1…H_t. The set of hash functions must be the same on both client and server machines. These hash functions are chosen to be one-on-one maps from the space of signature values to some well-ordered set of values. This generates t image sets, each of size n:

$IS_1 = \{H_1(Sig_1), H_1(Sig_2), \dots\}$
…
$IS_t = \{H_t(Sig_1), H_t(Sig_2), \dots\}$

Step T.2: The pre-traits PT_1…PT_t are computed by taking the signatures whose hash is the minimum element of each image set. Intuitively, using the minima means that if two files only have a few chunks that differ their pre-traits are likely to be the same.

$PT_1 = Sig_{1j}$ where $H_1(Sig_{1j}) = \min(IS_1)$;
…
$PT_t = Sig_{tj}$ where $H_t(Sig_{tj}) = \min(IS_t)$;

Step T.3: The traits T_1…T_t are computed by selecting b different bits out of each pre-trait PT_1…PT_t. For instance:

$T_1 = select_{0..b-1}(PT_1)$
…
$T_t = select_{(t-1)b...tb-1}(PT_t)$

In DFSR, to compute H_t on a given signature value we calculate the MD4 signature concatenated with the 1-byte representation of t. If the pre-traits have fewer than $t\times b$ bits, we could use an extra level of hash function to compute independent values, but in the range of values we consider experimentally, this does not happen. We want the bit ranges to be non-overlapping so that repeated values in the pre-traits (as may happen for short files) do not produce repeated values in the traits, which would allow accidental collisions of b bits for unequal signature values to be magnified.

The number of traits t and the trait size b are chosen so that just a small total number of bits ($t\times b$) is needed to represent the traits for an object. Typical combinations of (b,t) parameters that we have found to work well are (b=6,t=16) and (b=4,t=24), for a total of 96 bits per object. Abusing notation, we'll denote by $T_i(F)$ the i^{th} trait of F.

D. Computing the pre-traits efficiently

An efficient way to select the pre-traits PT_1 … PT_t in step T.3 is to pick an expanded signature set and to perform partial evaluation of the signatures. Logically, each H_i is divided into $High_i$ containing the high-order bits of H_i, and Low_i containing the remaining low-order bits. Since only the minimum element of each image set IS_i is selected, we compute $High_i$ for every signature, but need to compute Low_i only for those signatures that achieve the minimum value ever achieved for $High_i$. If the High values are drawn from a smaller space, this may save us computation. If, further, several High values are bundled together, we can save significant computation. Suppose, for instance, that each High value is 8 bits long. We can pack eight of these into a long integer; at the cost of computing a single random value from a signature, we can chop that value into eight independent one byte slices. If only the High value were needed, this would reduce our computational costs by a factor of eight; however, on average one time in 256 we also need to compute the corresponding Low value.

Because our trait values are just the input signature values, if we have only one signature which attains the minimum High value, we need never compute the corresponding Low value.

E. Finding similar files using a given set of traits

We can approximate the set of files similar to a given file F_S by computing the set of files having similar traits to F_S:

$$\mathrm{TraitSim}(F_S, F_C) = \left|\left\{i \mid T_i(F_C) = T_i(F_S)\right\}\right|$$

$$\mathrm{SimTraits}(F_S, \mathrm{Files}_C, n, s) = \left\{F_{C1}, F_{C2}, \ldots, F_{Ck}\right\}$$

where the set $\{F_{C1},\ldots,F_{Ck}\}$ contains $0 \le k \le n$ files in Files_C such that $(\mathrm{TraitSim}(F_S,F_{Ci})/t) \ge s$, and for all other files x in Files_C, $\mathrm{TraitSim}(F_S, x) \le \mathrm{TraitSim}(F_S, F_{Ci})$. Note that TraitSim is an unbiased estimator for Sim, if we divide by t, when averaged over all choices of one-to-one hash functions.

The calculated curves in Fig. 3 show the probability of detecting 1,2,…,16 matching traits as a function of the actual fraction of matching chunk signatures. Note that the similarity curves allow us to detect true similarity differences in the range of 5-10%. For the set of parameters (b=6,t=16) that we use in practice, we found that setting the lower bound to s = 5/16, i.e. at least 5 matching traits, provided a reasonable threshold. The false positive rate for entirely dissimilar files drops to roughly one in three hundred thousand at 5 out of 16 matching traits (giving us 30 bits of true match, but $\binom{16}{5}$ ways to generate these matches).

To efficiently determine the set of likely similar files, we organize our traits into $t \times 2^b$ lists of files indexed by (τ, β); in list (τ, β) we include every file in which trait τ has value β. We identify the files using small integers and delta-encode the lists for compactness. Given Traits(F_S), we can select the t lists corresponding to the trait values of F_S, and then perform the equivalent of a merge sort to compute the number of matching traits for every file in Files_C. We can maintain the $k \le n$ largest values as yet observed in a bounded-size priority queue.

Using this compact encoding, the trait information for 1M files can be kept in about 32MB of main memory. For (b=6,t=16), each bucket has on average 16K entries, so the merge takes about 96K comparisons, less than a millisecond of CPU time.

To improve both precision and recall, we could increase the total number of bits. For instance, switching to (b=5,t=24) would dramatically improve precision at the cost of increasing memory consumption for file traits.

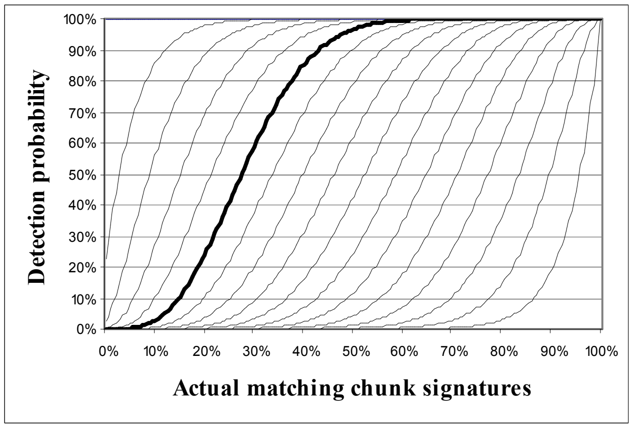

Fig. 3. Calculated similarity curves for (b=6,t=16) showing the detection probability of finding 1, 2, …, 16 matching traits, given the actual fraction of matching chunk signatures. The bold curve is the one used in DFSR, corresponding to 5 matching traits.

IV. Recursive RDC Signature Transfer

For large files, a fixed overhead is incurred in Step 4 of the basic RDC protocol described in Section II even if F_C and F_S are very similar. The amount of data sent over the network in Step 4 is proportional to the size of F_S divided by the average chunk size. This can become quite significant for large files. For instance, assuming the size of F_S is 10 GB and the average chunk size 2 KB, F_S will be divided into 5 million chunks, corresponding to about 60 MB of signature information that needs to be sent over the network in Step 4. This is a fixed cost, even if the differences between F_C and F_S (and thus the amount of data that needs to be sent in Step 6) are very small.

To reduce the amount of signature traffic sent over the network in Step 4, a recursive application of the basic RDC protocol can be used to transfer the signatures. Thus Step 4 of the basic protocol may be replaced with the following steps:

Step R.4.1: The ordered list of chunk signatures and lengths $((\mathrm{Sig}_{S1},\mathrm{Len}_{S1}) \ldots (\mathrm{Sig}_{Sn},\mathrm{Len}_{Sn}))$ computed in Step 3 is recursively chunked up on S into *signature chunks* using an approach similar to that described in Step 3, to produce a list of recursive signatures and lengths $((\mathrm{RSig}_{S1},\mathrm{RLen}_{S1}) \ldots (\mathrm{RSig}_{Sr},\mathrm{RLen}_{Sr}))$. Compared to the original list, the size of the recursive list is reduced by a factor equal to the average chunk size ($r \ll n$).

Step R.4.2: C also does a recursive chunking of its signature and length list $((\mathrm{Sig}_{C1},\mathrm{Len}_{C1}) \ldots (\mathrm{Sig}_{Cm},\mathrm{Offs}_{Cm}))$ into signature chunks, obtaining a list of recursive signatures and lengths $((\mathrm{RSig}_{C1},\mathrm{RLen}_{C1}) \ldots (\mathrm{RSig}_{Cs},\mathrm{RLen}_{Cs}))$, where $s \ll$

m.

Step R.4.3: S sends C its ordered list of recursive signatures and lengths $((RSig_{S1},RLen_{S1})\ldots(RSig_{Sr},RLen_{Sr}))$.

Step R.4.4: C compares the recursive signatures received from S with its own list of recursive signatures computed in Step R.4.2. C then sends a request to S for every distinct signature chunk (with recursive signature $RSig_{Sk}$) for which C does not have a matching recursive signature.

Step R.4.5: S sends C the requested signature chunks.

Step R.4.6: C uses its locally-matching signature chunks and the chunks received from S in Step R.4.5 to reassemble the original list of signatures and lengths $((Sig_{S1},Len_{S1})\ldots(Sig_{Sn},Len_{Sn}))$ computed by S in Step 3. At this point, execution continues at Step 5 of the basic RDC protocol described in Section II.

For very large files, the above recursive procedure may be applied *r* times ($r \geq 1$). For an average chunk size *C*, this reduces the size of the signature traffic over the network by a factor of approximately C^r.

The recursive step may be performed by choosing an average chunk size based on the number of signatures from the previous iteration. This approach may be used to bound the amount of data that gets sent over the wire. The method may be extended to also choose a permitted overlap, where the chunks may overlap with a number of common signatures. The recursive decomposition described above can be seen as an instance of this where the permitted overlap is 0 and the chunk size is set to *C*.

In contrast to LBFS [24], our experimental results suggest that, when using recursion, the highest bandwidth savings can be achieved by choosing a small average chunk size (e.g. *C*=2048 for the base and *C*=256 for the recursive levels). This observation is also consistent with the fact that recursion will save significantly on longer file segments that are equal, since the recursive signatures will coincide, while segments that are different are often relatively short, especially in text documents.

V. Improved Chunking Algorithms

In this section we discuss three alternative chunking algorithms and introduce a metric, called *slack,* for comparing these. Calculations that are too lengthy to fit here are reported in [2].

A. Slack and using Point filters

We observed that the basic RDC protocol requires the client and server to partition their files independently. A technique for achieving this, described in [24], pre-processes the file stream through a hash that summarizes the contents of windows comprising the last *w* bytes (typically *w*=12...64) into a single 4-byte value *v*. The Rabin hash [28] provides a high quality hash function that can be computed incrementally. Choose a number *h* and identify as cut points the positions where $v \bmod h = 0$.

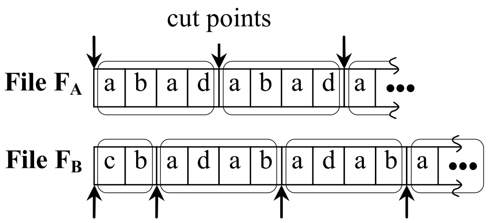

Fig. 4. Example of misaligned cut points for *m*=3 and *h*=a when using the point filter method, where *m* is the minimum chunk size and *h* is the value at which cut-points are chosen.

There is no gain to sending chunk signatures that occupy more space than the chunks they summarize. Therefore, to filter out cut-points that are considered too close, one can supply an additional parameter *m* that indicates how many positions should be skipped before *v* is checked again against *h*. This method produces average chunks of size *m*+*h*.

How should *m* and *h* be chosen? If *m* is large relative to *h,* then there is a high probability that chunks in F_S and F_C will not align even though F_S and F_C only differ in the first character. In Fig. 4 we illustrate the situation in which cut points are misaligned, for a choice of parameters *m*=3 and *h=a* (for simplicity, we have suppressed the use of a rolling hash in this example).

Such misalignment directly influences the overhead of RDC, since the amount of file data transferred by RDC is directly proportional to the number of chunks of F_S, but not of F_C. For quantitative comparisons of chunking methods, we therefore consider the following scenario. Take two doubly infinite files f_1, f_2 which coincide on non-negative positions. These non-negative positions form an infinite-to-the-right file f_3. Let f'_1 (respectively f'_2) be the part of f_1 (respectively f_2) given by their respective negative positions. Assume that the infinite-to-the-left files f'_1 and f'_2 as well as the infinite-to-the-right file f_3 are random. We then define the random variable *slack* as the distance from 0 to the least position of f_3 that is the common cut point of f_1 and f_2, normalized by dividing by the expected chunk size.

Since bytes “wasted” in the slack will be in chunks whose signatures don’t match, RDC benefits from a chunking method with the smallest possible expected slack. For the *point-filter* method it can be shown that the minimal slack (obtained for $m = 0.3h$) is ≈ 0.82.

B. Using an interval filter

A different way to choose cut points is what we will refer to as the *interval-filter* method. It determines the next cut-point by searching for a pattern in an interval of *h* previous values. It avoids indefinite misalignments. A prototypical

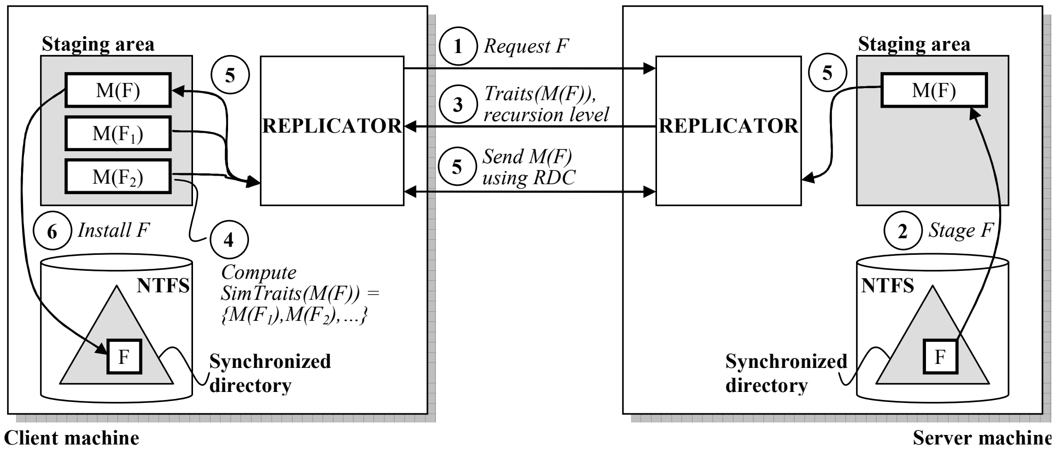

Fig. 5. Replicating a file F from a server to a client involves marshaling the file F and its metadata into a staged representation M(F) on the server, negotiating the traits and the RDC recursion level, identifying a set of similar staged files M(F_1), M(F_2),… on the client, using RDC to transfer M(F), and finally un-marshaling the transferred M(F) into a copy of the file F on the client.

example of an interval filter is obtained by partitioning the hash values v into two sets, H (head) and T (tail), where H contains $1/h$ of the possible hash values and T contains the rest. A cut-point is then chosen when the file matches a pattern of h-1 T's followed by one H. Clearly, cut-points are at least h positions apart, and the choice of possible cut-points is local, since it depends only on the h previous positions.

The average chunk size when using an interval filter can be shown [2] to be $e \cdot h$, while the slack is $1 - e^{-1} + e^{-2} \approx 0.77$.

C. Using local maxima

The point and interval filters are both absolute in the sense that they require hash values to take certain pre-determined values for a position to be a cut-point. Another local way, closely related to the winnowing technique described by Schleimer [30], is to choose as cut-points the positions that are *local maxima* (or local minima). An h-local maximum is a position whose hash value v is strictly larger than the hash values at the preceding h and following h positions. Suppose there are M different possible hash values ($M=2^{32}$ for 4-byte hash values), then the probability that a given position is a cut-point is:

$$\sum_{0\le j\le M} \frac{1}{M}\cdot\left(\frac{j}{M}\right)^{2h} \approx \frac{1}{2h+1}$$

since, for each of the M different values that a position can take, the neighboring $2h$ positions must be taken from the j/M^{th} fraction of smaller values. Kac's ergodic recurrence theorem [26] implies that the average distance between cut-points is the inverse of the probability: 2h+1. The expected slack for the local maxima filter turns out to be ≈ 0.7.

One intriguing observation is that, on average, the queue used to store the ascending chain of at most h previous hash values will have length ln(h). To see this, denote by $f(h)$ the expected length of the sequence starting with the hash at the current position and including hash values from at most h preceding positions that form an ascending chain. In the base case, we have $f(0)=0$, while in general, the current position is included and the next position to be included is taken uniformly from the remaining h-1 positions, unless the value at the current position is maximal. The h^{th} harmonic number is the solution to the recurrence equation:

$$f(h) = 1 + \frac{1}{h}\sum_{i=0}^{h-1} f(i) = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{h} = H_h \approx \ln(h)$$

We can use the above observation to compute cut points by examining every position only 1+ ln(h)/h times on average. The algorithm processes chunks of size h. Each chunk is processed from right to left building up an array of strictly ascending hash values. The largest value in the *k+1'st* interval is marked maximal if the first dominating value in the k^{th} interval is beyond h positions away. On the other hand, the largest value in the k^{th} interval is a cut-point if it is marked maximal, and is either larger than the largest value in the k+1st interval, or the largest value is beyond h positions away, and all values closer to it are smaller.

As we later report on in Section VII.A, there is a performance tradeoff between the different chunking methods. The average case behavior of only *ln(h)* branch miss-predictions together with the ability to use scanned bytes directly as digits[1] appear to give local maxima an both a performance and quality advantage over point filters that rely on a good hashing function. On the other hand, local maxima require a look-ahead for determining cut-points that makes it harder to compose with other stream processing utilities.

[1] As far as we know, this simple fact appears to have not been observed before. An efficient implementation of this approach resembles a Boyer-Moore string matching algorithm.

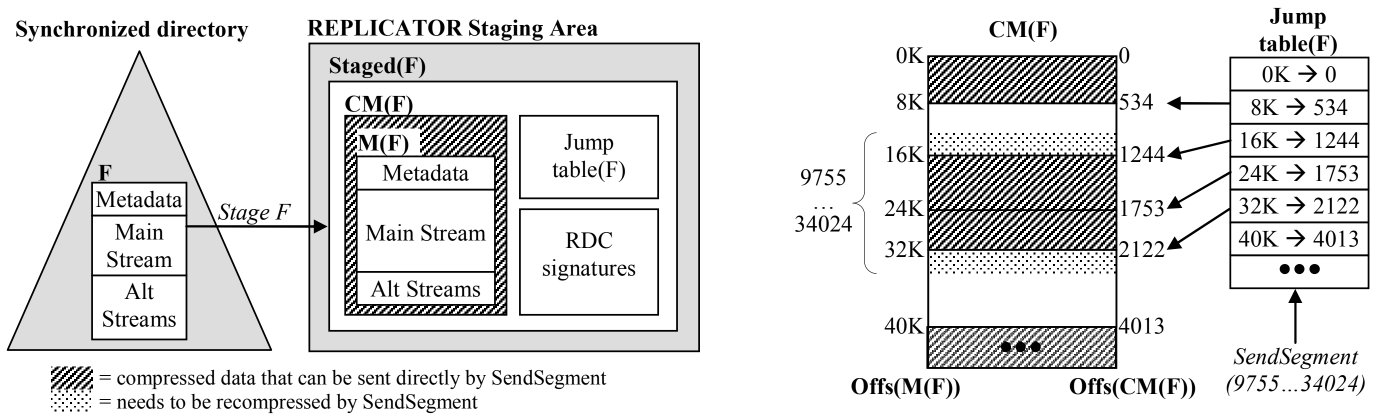

Fig. 6. The staged representation of a file F contains the marshaled file metadata and data streams M(F) stored in compressed form CM(F), the RDC signatures for M(F), and the jump table for CM(F). The jump table is used for seeking inside CM(F) given logical offsets into M(F), and is based on storing the offsets in CM(F) for every 8K segment in M(F); seeking inside a segment requires decompression of the segment.

In certain applications, it may be useful to also impose a maximal size on chunks. None of the considered chunking methods impose maximal sizes a priori, as it is impossible for methods that depend only on a limited neighborhood to impose bounds on both minima and maxima (consider a file of all 0's: either all positions are cuts or none are). On the other hand, periodically imposing extra cuts based on fixed lengths does not break RDC. Obviously, local max can be refined in either direction by allowing cuts at positions that are h' maximal, for $h'<h$, (or $h'>h$ as part of recursive RDC).

VI. Implementation in DFSR

This section details the implementation of our previously described RDC approach in the DFSR service included in Windows Server 2003 R2. Given that an in-depth description of DFSR is outside the scope of this paper, we shall focus on how files are transferred between machines using RDC; a general description of state-based replication can be found in Saito and Shapiro [29].

DFSR uses an algorithm based on version vectors to efficiently determine which files of a synchronized directory tree need to be replicated between a server and a client machine[2]. Once the client has found out that a file F needs to be replicated, the following steps are executed to differentially transfer F, as illustrated in Fig. 5:

Step F.1: The client sends a request to the server to prepare for the differential transfer of F.

Step F.2: The server marshals the data and meta-data of F into a blob M(F) and stores this blob in a private directory called the *staging area*. For NTFS files, M(F) needs to include the main data stream of F, any alternate data streams, as well as the replicated meta-data such as the file creation time and ACL (access control list); some of the metadata, such as the last modified time, is not replicated and is thus not included in M(F). Based on the size of M(F), the server determines the RDC recursion level, and computes the RDC signatures and the traits for M(F). Transfers of files smaller than 64K fall back to direct downloads without RDC.

Step F.3: The server sends Traits(M(F)) and the desired RDC recursion level to the client.

Step F.4: The client uses its similarity information to compute SimTraits(M(F)) = {M(F_1), M(F_2), …}, thereby identifying a subset F_1, F_2, … of its own files whose marshaled representations M(F_1), M(F_2), … are similar to M(F). For each of these files F_k, the client computes M(F_k) and stores it in its staging area. To bootstrap the use of RDC, the client also considers pre-existing files with no traits stored in the similarity information, but with the same name as the remote file.

Step F.5: M(F) is transferred via recursive RDC (as described in Section IV) using M(F_1), M(F_2), … as a seed.

Step F.6: Once the client has reassembled a copy of M(F) in its staging area, it un-marshals it to obtain its own copy of F, which it installs in the synchronized directory.

Computing the traits and RDC signatures of M(F) instead of F has the advantage of enabling RDC to also work for changes to the file meta-data. For instance, this occurs when the NTFS ACLs for all files in a directory tree are changed recursively.

DFSR manages its staging area as a cache. Staged files are deleted lazily when the staging area size reaches a

[2] As mentioned in the introduction, the terms "client" and "server" only refer to the direction in which synchronization is performed. A machine can act both as a client and a server at the same time. For instance, this is the case for bi-directional synchronization between two machines.

configurable threshold. If M(F) is cached, a machine acting as a server may thus skip step F.2, while a machine acting as a client will not need to re-compute M(F_k) if it decides to use F_k as a seed in step F.4.

Should the differential transfer of M(F) be interrupted in step F.5 (e.g. because of a broken connection to the server), the partial download of M(F) is kept in the client's staging area and will be reused as a seed during future attempts at transferring M(F). This provides the equivalent of download resumption at very low additional cost, and allows downloads to be reliably resumed even across different server machines.

Given that the synchronized data that we typically see in production systems tends to have a reasonably good compression ratio, it is possible to further reduce bandwidth by transferring compressed chunks if no pre-existing similar data can be identified. However, both compression and RDC signature computation incur noticeable CPU overhead.

To reuse the result of compression and RDC chunking, we keep a staged representation of F, Staged(F), in the staging area instead of just storing M(F). Ass shown in Fig. 6, Staged(F) includes the *compressed* marshaled representation CM(F) and the RDC signatures for F. Staged(F) is stored as a single NTFS file, where the unnamed data stream contains CM(F), while alternate NTFS file streams contain the RDC signatures (one stream per recursion level) and a jump table that we describe next.

Since seeks are required for steps S.7 and S.8 of the RDC protocol to retrieve file chunks on both client and server, we have adapted the compression algorithm to allow us to perform reasonably efficient seeks on the compressed format CM(F). This is done by compressing 8K segments of M(F) at a time and maintaining a *jump table* consisting of an array of offsets into the compressed stream for each of the 8K segments, as illustrated in Fig. 6. A lookup in the jump table consists of dividing the required offset by 8K and reading the resulting position in the array to get to the surrounding compressed block containing the desired offset. When transferring chunks over the wire in step S.7 of the RDC protocol, we use the jump table to avoid re-compressing portions of CM(F). For instance, referring to Fig. 6, when serving the range from uncompressed offset 9755 to 34024, only the portions that don't fit within an existing segment (9755…16383 and 32768…34024, respectively) are re-compressed by the server's *SendSegment* routine. Portions corresponding to whole segments can be transferred to the client directly out of CM(F).

While staging is beneficial when the same file is served to multiple clients or when a file is used as a seed, maintaining a staging area comes at the cost of storage and disk access overhead. To mitigate this cost, we skip the creation of Staged(F) when F is smaller than 64K and RDC is not used.

A memory mapped file stores the similarity information described in Section III. Since this data structure takes up about 32 bytes per file, most of this information can be cached in memory even for synchronized directories with a large number of files. In addition to the *16 × 2^6* lists of file IDs (for which most entries can be represented using only one byte due to delta-encoding), we need to maintain a file ID table that maps the compact file ID to the UID (unique identifier) assigned by DFSR to the file.

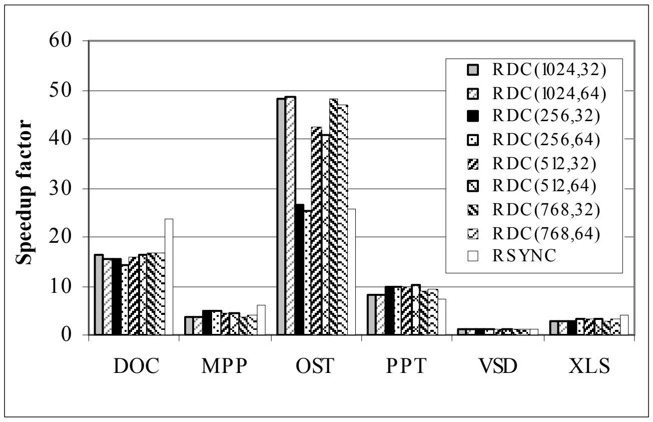

Fig. 7. RDC speedup factors without recursion, using various combinations of the horizon *h* and window size *w*, compared to RSYNC. The average chunk size is *C*=2*h*+1.

Updates to the similarity information are periodically flushed to disk and the similarity file is marked on clean shutdowns. Additionally, whenever we compute or receive the traits for a file, we store them in the database used by DFSR to keep track of file UID and version numbers. If the similarity file is lost or corrupted (e.g. because of a dirty shutdown), it is rebuilt on startup using a scan of the records in the DFSR database. While discrepancies between the similarity file and the actual content of the synchronized directory do not impact correctness, they have a performance cost in terms of lost RDC opportunities.

To compute the list of needs in Step S.5 of the RDC protocol, the client inserts the stream of remote signatures received from the server in Step S.4 into a hash table that maps signature hashes to file offsets in the server's file. The client then scans its seed signatures stored in Staged(F), performing a lookup in the hash table to establish whether a match exists for each signature. If the size of the hash table is too large to fit in memory, the client processes the remote signatures in batches, doing a sequential scan of the signatures in Staged(F) for every batch We found that using a hash table was approximately 50% faster than using binary search in a sorted list, most likely due to the good distribution of the signature hash function.

VII. Experimental Results

In this section we present experimental results obtained by using DFSR, and compare these results against an LBFS-like approach [24], RSYNC [33][34], and the local diff utilities xdelta [21] and BSDiff [25]. To obtain the LBFS figures, we implemented a simulator that allows us to determine the transfer sizes for various experiments without actually providing the full file system semantics of LBFS. All network traffic reported below refers to application data and does not include network overhead such as IP packet headers.

By default DFSR compresses all data sent over the wire using a proprietary compression library. Since our compression algorithm has different characteristics than zlib used in RSYNC, and since we're using blocked compression, most of the figures in this section are for uncompressed traffic to provide an accurate comparison.

Hardware characteristics are indicated where relevant for performce, otherwise, our measurements in terms of bytes over the wire are hardware agnostic. DFSR was run on a Windows Server 2003 R2 installation, and RSYNC was run on RedHat Linux 2.4.

A. *Reducing the overhead of computing chunks*

Early performance testing indicated that calculating chunk boundaries and chunk signatures contribute significantly to the overall CPU overhead of RDC. Consequently, we hand-optimized the most CPU intensive parts by using assembly versions of critical inner loops and MD4 and achieved very significant speedups.

To compare the optimized chunker against RSYNC and the local diff utilities, we measured the chunking and signature comparison overheads combined and aggregated the client and server overhead for RDC and RYNC. On a P3 machine, and for identical files, we measured 31 cycles per byte for RDC, 45 for RSYNC, 39 for xdelta, and 2580 for BSDiff. When using a pair of different files, we measured 36 for RDC, 32 for RSYNC, 410 for xdelta and 2780 cycles for BSDiff. Thus, RDC and RSYNC appear comparable, while the local diff utilities require much more CPU and memory.

The optimized RDC chunker requires 31 CPU cycles per byte on a Pentium 4 (corresponding to 64MB/s throughput on a 2GHz processor), and 24 cycles per byte on an AMD64 CPU running x86 binaries (75MB/s on a 1.8GHz processor).

When DFSR sends a file to more than one client, the CPU overhead for chunking is amortized by persisting the chunks and signatures in Staged(F) on the server, as described earlier. In contrast, RSYNC spends most of the CPU cycles for file transfers on the server, as the chunking and the lookup on the server are dependent on the client data.

More recent experiments indicate that a 128 bit hash based on Jenkins' hash algorithm [17] is adequate and twice as fast as MD4. Furthermore, another significant speedup can be gained by computing local maxima directly by treating the bytes from the input file as digits in large numbers. This allows bypassing computing rolling hashes all-together. For instance, for an average chunk length of 256 bytes, a 64 bit P4 machine, requires 8.4 cycles per byte to compute local maxima when bypassing the rolling hash, but 18.7 cycles per byte when layering the computation with a simple, low quality, rolling hash based on bit-wise exclusive or and bite-wise rotation. The point-filter approach, on the other hand requires a hash, but has lower overhead when determining cut-points. Hence, for the low quality hash, it requires only 7.6 cycles per byte; but for the higher quality Rabin based hash we measured 15.8 cycles per byte.

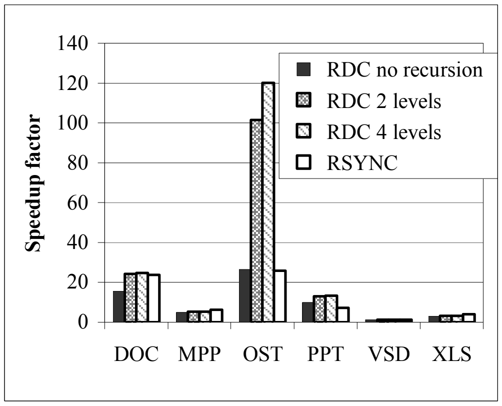

Fig. 8. RDC speedup factors for a fixed horizon and window size (*h*=256, *w*=32) but with variable levels of recursion, compared to RSYNC. The biggest boost is achieved for large files.

B. *Tuning the chunking parameters and the recursion level*

In the next set of experiments, we first tune the chunking parameters by choosing a horizon and window size *h* and *w*, respectively. We examine the impact of recursion next, and evaluate the bandwidth savings against RSYNC and the local diff utilities. The data set used for these experiments consists

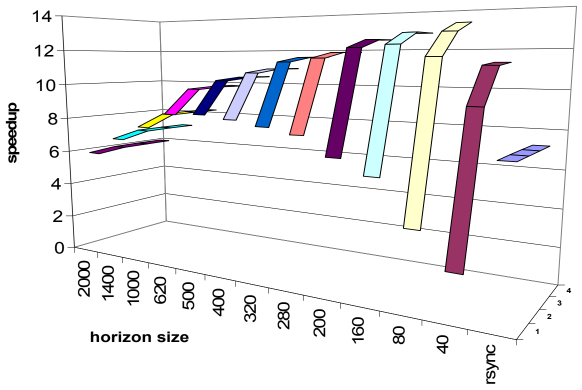

Fig 9. Speedup based on a sample pair of PPT files, as a function of horizon size (sampled between 40 and 2000) and recursion level (between 1 and 4).

of several different file types: Word (DOC, avg. size: 19MB), PowerPoint (PPT, 9.2MB), Excel (XLS, 6MB), Visio (VSD, 2MB) and Project (MPP, 1MB); for each of these types, the seed files were 15 randomly selected documents from an archive. The data set also includes two Outlook Offline Folders (OST) files. The changed files were derived by editing one or two places within each file; the edits included changing a few headers, word replacements, font changes, additions or deletions of text, and changes in diagrams. For the OST files, we used versions from successive days (updated with a few hundred emails) as the changed files.

Fig. 7 shows the RDC speedup factors achieved for various combinations of the horizon *h* and window size *w* (remember that for local maximum the average chunk size is C=2h+1). The speedup factor is calculated as the total size of the file divided by the number of bytes sent. Fig. 7 shows weighted averages across samples of the same type. Even without recursion, RDC compares quite favorably to RSYNC, though the latter is better in several cases because it sends fewer and more compactly represented signatures. This is the reason why RDC doesn't perform as well on OST or DOC files for a small average chunk size and no recursion.

In Fig. 8 we fix the horizon and window size to (h=256, w=32) and vary the number of recursion levels (0, 2, and 4). We picked a small average chunk size since the additional signature overhead is reduced through recursion. The parameters for the recursive signature chunking were (h=128, w=4). As shown in this Figure, recursion provides a very significant performance boost, especially for larger files: while the average PPT file size in our experiments was 8.8MB, the OST files were about 213MB in size.

Fig. 9 summarizes an experiment performed on a PPT file selected at random. A combination of horizon size between 80 and 160 and 4 recursive applications of RDC provide the highest speedup (about 13x), while recursion has little effect for the higher horizon sizes.

Experiment	File size	RDC speedup	RSYNC speedup
Baseline: VHD containing a clean installation of Windows Server 2003 SP1.	3.2GB	n/a	n/a
Add MSN messenger and toolbar .	3.2GB	104	36
Add MSN money.	3.8GB	92	2
Add Winzip and Source Insight.	3.8GB	82	73
Install R2 and launch DFSR.	4.0 GB	13	15
Baseline: Windows Server 2003 Domain Controller VHD image.	2.5GB	n/a	n/a
Add new domain user account.	2.5GB	886	439
Install the admin pack.	2.6GB	236	162
Add another domain user account.	2.5 GB	1036	439

Table. 11. Speedup factors for RDC vs. RSYNC for large files.

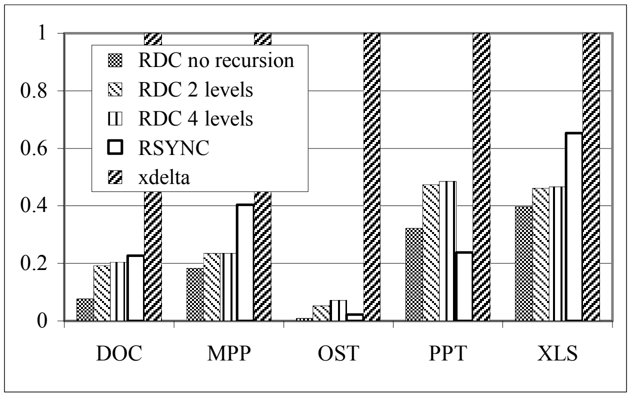

Fig. 10. Speedup of recursive RDC and RSYNC normalized to xdelta. Note these figures all include compression.

To calibrate how well RDC and RSYNC compare to local diff, we ran the same dataset with xdelta. We expected xdelta to perform better, since it has more information available to it than RDC or RSYNC. Note that xdelta uses significantly more memory than RDC (8 times the total file sizes for xdelta, vs. less than 5% of the file size for RDC if the signatures are kept in memory). The results are shown in Fig. 10. Note that for this comparison we had to use compression, since the output of xdelta is compressed. In some cases our blocked compression does not perform as well as the one in xdelta or RSYNC, which is applied to the entire change stream. This was a trade-off we made in favor of reducing the CPU overhead on servers that need to replicate data to large numbers of clients.

To further examine the impact of recursion on very large files, we ran RDC and RSYNC on some VHD (Virtual Hard Disk) files generated using Microsoft Virtual Server. Table 11 summarizes the experiments we performed. The baseline was a Windows Server 2003 clean install image. In the first four experiments we used the result of the updates listed in the first column as the initial VHD version for the next row. Before each experiment, we scrubbed the VHD files by zeroing out all unallocated sectors and deleting the OS page file to avoid transferring dead sectors. The results are shown in Table 11, note: we could not run the local diff utilities on these files, as their memory consumption far exceeded the available main memory. RDC was run with 4 levels of recursion, and a horizon size of 256 bytes. The first six rows correspond to the six experiments listed above. The last row repeats the firth experiment, but uses the admin pack installation as an additional seed. Notice how recursion plays a significant role when the speedup factor is high, while RSYNC is on par with RDC on a more modest speedup factor.

C. *Evaluating similarity detection*

We now present the results from a set of experiments in

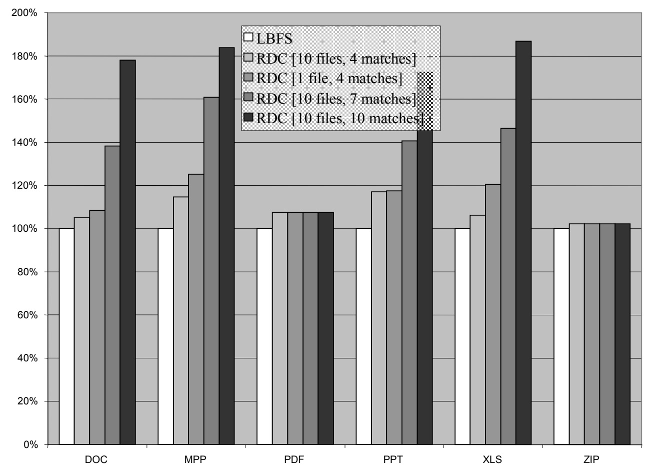

Fig. 12. Comparison of the transfer bandwidth required when using similarity detection with different parameters and RDC (without recursion) vs. LBFS.

which we tuned similarity detection and compared the results against LBFS.

The file data used was drawn from a versioned document library maintained by our organization. We selected at random a total of 14,000 old versions of files from the library to be used as seeds. We then picked the new versions of 350 of those files, which included various Microsoft Office formats, as well as PDF and ZIP files. For each new file, we computed the traits and ran similarity detection to select *n* similar files, by using *k* matches Next, we used the *n* similar files for an RDC transfer (without recursion) from a server containing the new file to a client containing the 14,000 seed files, and measured the required bandwidth. We normalized the results against the bandwidth required for an LBFS transfer. The outcome is shown in Fig. 12 for samples where $n \in \{1,10\}$ and $k \in \{4,7,10\}$. We used 16 traits with 6 bits each in all experiments reported in this table.

Another experiment used a collection of 36 VHD images (using 24 as seeds), totaling 100GB. The data set included VHD images of different versions of operating system images, ranging from MS DOS to Windows Server 2003 images. The choice of the parameters *k* and *n* had virtually no impact, as for each transferred VHD only one seed contained most of the useful data for RDC and LBFS. So RDC was within 10% on all transfers without recursion. Adding recursion turned the advantage 10% towards RDC, but not more, as the data set consisted of larger deltas than used in Table 11.

The results in Fig. 13 show that for a choice of parameters of (n=10,k=4), the most liberal matching criterion, our technique performs within just a few percent of LBFS in most cases, and within 10% for the OS images. This is remarkable, given that the amount of metadata we keep and our overhead in terms of potential disk seeks per chunk are tiny compared to LBFS, and that LBFS has ideal chunk reuse characteristics. The advantages using similarity are furthermore amplified by recursion and small chunk sizes.

Type	Server 1	Server 2	Server 3
Total transfers	219061	175854	160926
Non-RDC transfers	195623	156555	148625
Non-RDC bytes transferred	2.9GB	1.4GB	6.5GB
RDC transfers	23438	19299	12301
RDC target size	18GB	15GB	8.4GB
RDC bytes transferred	13.5MB	118MB	238MB
Average speedup from RDC	133	128	35

Table 13. Breakdown of RDC and non-RDC transfers in a deployment scenario.

D. Deployment example

A scenario that illustrates a realistic, albeit not controlled, use of RDC is a DFSR deployment among 136 globally distributed branch offices within Microsoft. DFSR is used to replicate product builds as well as documents to the branch offices. While the total amount of replicated data is currently around 100GB, churn only happens occasionally when updates such as patches are shared, ACLs are changed, and old content is removed. To gauge the amount of data that may be transferred and the contribution of RDC, we sampled the activity of three servers over a week. A breakdown is summarized in Table 13. We observe that the average file sizes for non-RDC transfer is around 30K and for RDC the average file size is around 750KB, while on average only 10KB of actual changes were transferred by RDC.

The very significant speedups from Table 13 may suggest that only trivial churn appears on the machines. With additional instrumentation, we extracted a summary of which heuristic DFSR used to identify similar files for RDC. Besides the similarity metric, DFSR also allows using partially downloaded files, old versions of the same file, and name conflicting files as a seed for RDC. Table 14 summarizes the collected numbers. The first column contains the combination of heuristics used for a transfer, the number of times that particular heuristic was used is counted in the second column, and the combined file sizes are summarized in the third column. While old versions of the same files account for a significant amount of the applicable heuristics, we observe that there is a very significant presence of similar files. In the Microsoft scenario this is due to a relatively frequent case of duplicate files across different replicated folders.

Type	Count	File sizes	Speed up
Total RDC downloads	21063	126MB	89
Old version + 1 similar file	11229	53MB	86
Old version	3498	16.5MB	199
Name conflict + old version	1432	8.8MB	85
Partial download + old version + 1 similar file	1223	5.4MB	93
2 similar files	1139	17MB	25
Old version + partial download	1087	7.4MB	121
Name conflicting file	748	6MB	72
3 similar files	210	1.4MB	52
Other combinations	495	10.6MB	30

Table 14. Breakdown of RDC seed heuristics.

VIII. Related Work

Several previous papers have been concerned with the efficient identification of similar objects in large collections. Manber [22] investigated the use of probabilistic selection techniques for finding local files closely resembling a given file, to help with version management. In order to ascertain the degree of near-duplication of pages on the World-Wide Web, Broder et al. [3][4] employed similarity techniques from which the one we present is derived. Heintze [11] and Schleimer et al. [30] apply related techniques with a primary focus on discovering duplication of code fragments for detecting plagiarism. The latter paper additionally presents *winnowing*, a phrase-selection scheme based on local minima of hash functions, algorithmically close to the one we use in defining chunk boundaries, apart from the treatment of ties A central result of [30] is that the winnowing has a density of selection within a constant factor of the lower bound of any landmarking scheme. In contrast to our criteria, winnowing requires at least one chunk boundary for every segment of size *h*. Consequently, no minimal chunk length is or can be enforced by winnowing. On the technical side, our contributions include the notion and analysis of slack, the average number of branch miss-predictions, and the simple observation that no rolling hash is required for our schemes.

Cox [6] presents a similarity-based mechanism for locating a single source file, used to extend an RDC mechanism similar in most other ways to LBFS [24]. Irmak et al. [14] present an intriguing approach for reducing communication rounds in an RDC approach by using erasure codes; this improves latency at the expense of a small overhead in bandwidth. Suel et al. [31] consider using global techniques for discovering similar chunks, and then apply local differencing techniques. Kulkarni et al. [19] perform trace-driven analyses of differencing techniques for eliminating redundancy. Quinlan and Dorward [27] describe a protocol resembling LBFS, but at the disk block level. Korn et al [17], in their RFC, provide a network protocol to express the local edits needed in delta-encoding.

The widely used RSYNC protocol [33][34] has the recipient chop up its old file at fixed chunk boundaries. The recipient then transmits a strong and a weak checksum for each chunk to the sender. The sender traverses its version of the file, computing weak checksums over a sliding window. The weak checksums are used to filter out matching candidates with the chunks received from the recipient. The sender can then deduce which chunks already reside on the recipient and what file data needs to be transferred directly.

Langford [20] considers recursive decomposition as an extension to RSYNC. This decomposition, like that of Fu [9] and Irmak and Suel [15] (which both apply to new transport protocols, rather than RSYNC), constructs a balanced binary tree of segment fingerprints. The primary disadvantages of this approach are that the depth of the tree is larger than ours and is not tunable, and that small changes can cause misalignments throughout the tree. Consider, for example, a file containing 2^k chunks. Move the first chunk to the end of the file. All the leaf signatures are unchanged, but every signature at the next level of the tree and all higher levels is different. In the multi-round work of Langford [20], at least RSYNC alignment should apply, requiring computation of larger misaligned checksums to look for the transmitted signatures. In our design, small changes impact a small number of boundaries, and thus the set of chunks concatenated during the recursion will be largely unchanged.

Jain et al [16] present a replica synchronization system called TAPER that combines RSYNC for intra-file compression and LBFS for inter-file redundancy elimination. Directory renames are optimized by maintaining a hierarchical hash of directories. In contrast, DFSR maintains unique object identifiers per resource making renames cheap. TAPER makes a novel use of Bloom filters for content dependent hashes for similarity detection. A claim is made that Bloom filters are cheaper than techniques based on min-wise independent hashes and shingles. In contrast, Section III.D describes how the cost of computing min-wise independent hash functions could be reduced to less than one extra hash computation per chunk.

In a different setting, Chan and Woo [5] use related techniques to optimize the transmission of Web pages, building on chunks already resident in a cache.

The computational framework we use for computing similarity is derived from those of Broder et al. [3][4] and Fetterly et al. [8], with modifications to reduce memory usage and to locate several closely matching files.

Work that introduces concepts related to applying hashing recursively, but with substantially different content includes Eshghi [7] that uses a tree of hash values to represent a directory tree, and contains a suggestion using a two-level decomposition into chunks for P2P file copying. It suggests using a tree to represent versioning of files, but lacks details as to how the system determines appropriate-sized shared chunks (but different edits based on the same version of a file would give rise to a tree). It is based on Merkle [23] trees that are used for computing hashes in order to prove that a segment exists in the tree.

IX. Conclusion

We have presented three significant optimizations to previous work on remote differential compression protocols: a very efficient similarity detection technique, recursive signature transfer, and improved data chunking algorithms. These optimizations have been implemented in a commercial state-based multi-master file synchronization service that can scale up to a few thousand nodes.

Experimental data shows that these optimizations may help significantly reduce the bandwidth required to transfer file updates across a network, compared to previous techniques. Our similarity detection approach is shown to perform almost as well as the one used in LBFS (which has an ideal behavior in terms of chunk reuse), while requiring a very small amount of metadata per file (96 bits) and completely eliminating a substantial system-wide database of all chunks.

We showed that recursion plays a key role for transferring incremental differences between large files, such as Virtual PC images. The built-in minimal chunk size, the reduced average slack, and independence of rolling hashes are compelling reasons for using the local maxima chunking algorithm.

Some of the open issues that could be topics for future research include determining whether an *optimal* chunking algorithm exists with respect to slack, and applying RDC to compressed files.

X. Acknowledgments

We are thankful to Akhil Wable for developing and experimenting with an early prototype of RDC and similarity based traits. Eric Hammerle tuned our current implementation significantly and provided the CPU measurements. We are also indebted to Le Wang and Rob Post for stressing RDC from A to Z, and to Shobana Balakrishnan, Dan Boldo, Patrick Bozeman, Jeff Carollo, Richard Chinn, Brian Collins, Stone Cong, Huseyin Dursun, David Golds, Huisheng Liu, Ram Natarajan, Rafik Robeal, Christophe Robert, Masood Siddiqi, and Guhan Suriyanarayanan and numerous others for their drive for shipping RDC and DFS-R.

References

[1] M. Ajtai, R. Burns, R. Fagin, D. D. E. Long, and L. Stockmeyer, "Compactly Encoding Unstructured Inputs with Differential Compression," *Journal of the ACM, Vol. 49, No. 3,pp. 318-367,*5- 2002.

[2] N. Bjørner, A. Blass, Y. Gurevich. "Content Dependent Chunking for Differential Compression. The Local Maximum Approach," *MSR Technical Report,* 12-2006.

[3] A.Z. Broder, "On the resemblance and containment of documents," *Proceedings of the Compression and Complexity of Sequences*, 1997.

[4] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig, "Syntactic clustering of the Web," *Proceedings of the 6th International Conference on WWW*, 9-1997.

[5] M.C. Chan, and T.Y.C. Woo, "Cache-based Compaction: A New Technique for Optimizing Web Transfer," *Proc. of the IEEE Infocom Conference*, 1999.

[6] L. Cox, C.D. Murray, and B.D. Noble, "Pastiche: Making Backup Cheap and Easy," *5th Symposium on Operating System Design and Implementation*, 12-2002.

[7] K. Eshghi: Intrinsic References in Distributed Systems. 675-680. *22nd International Conference on Distributed Computing Systems, Workshops (ICDCSW '02) July 2-5, 2002, Vienna, Austria, Proceedings*

[8] D. Fetterly, J. Weiner, M. Manasse, M. Najork, "A large-scale study of the evolution of Web pages," *Software – Practice and Experience*, Vol. 34, No. 2, pp.213-37, 2004.

[9] K. Fu, F. Kaashoek, and D. Mazières, "Fast and secure distributed read-only file system," *OSDI-4*, 10-2000.

[10] T. Haveliwala, A. Gionis, and P. Indyk. "Scalable Techniques for Clustering the Web", In *Proceedings of WebDB*, 2000.

[11] N. Heintze. "Scalable document fingerprinting." *1996 USENIX Workshop on E-Commerce*, November 1996.

[12] J.W. Hunt, and M.D. McIllroy, "An algorithm for differential file comparison," *Computer Science Technical Report 41*, Bell Labs, 6-1976.

[13] J.W. Hunt, and T.G. Szymansky, "A fast algorithm for computing longest common subsequences," *Communications of the ACM* 20(5):350-353, 5-1977.

[14] U. Irmak, S. Mihaylov, and T. Suel, "Improved Single-Round Protocols for Remote File Synchronization," *IEEE Infocom Conference*, 3-2005.

[15] U. Irmak, and T. Suel, "Hierarchical Substring Caching for Efficient Content Distribution to Low-Bandwidth Clients," *14th International WWW Conference*, May 2005.

[16] N. Jain, M. Dahlin, and R. Tewari. "TAPER: Tiered Approach for eliminating Redundancy in Replica Synchronization," *4th Usenix Conference on File and Storage Technology*, FAST 2005.

[17] R. Jenkins. "Hash Functions for Hash Table Lookup", http://burtleburtle.net/bob/hash/evahash.html, 1995-1997.

[18] D. Korn, J. MacDonals, J. Mogul, and K. Vo, "The VCDIFF Generic Differencing and Compression Data Format," *RFC 3284*, 6-2002.

[19] P. Kulkarni, F. Douglis, J. LaVoie, and J. M. Tracey, "Redundancy Elimination within Large Collections of Files," *Proceedings of the 2004 USENIX Annual Technical Conference,* Boston, MA, 6-2004.

[20] J. Langford, "Multiround Rsync," *unpublished*, 1- 2001.

[21] J.P. MacDonald, “File System Support for Delta Compression,” *Master’s Thesis*, UC Berkeley, http://www.cs.berkeley.edu/~jmacd.
[22] U. Manber, “Finding Similar Files in a Large File System,” *Technical Report TR 93-33*, Department of Computer Science, Univ. of Arizona, Tucson, 10-1993.
[23] R. C. Merkle. "A Digital Signature Based on a Conventional Encryption Function." In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, pp. 369--378, 1987.
[24] A. Muthitacharoen, B. Chen, and D. Mazières, “A Low-bandwidth Network File System,” *Proceedings of the 18th SOSP*, Banff, Canada, 10-2001.
[25] C. Percival, “Naïve Differences of Executable Code,” *Draft Paper*, http://www.daemonology.net/bsdiff.
[26] K. Peterson. “Ergodic Theory,” *Cambridge University Press*, 1983.
[27] S. Quinlan, and S. Dorward, “Venti: a new approach to archival storage,” *Proceedings of the FAST 2002 Conference on File and Storage Technologies*, 1-2002.
[28] M. Rabin. “Fingerprinting by random polynomials”. *Report TR-15-81*, Center for Research in Computing Technology, Harvard University, 1981.
[29] Y. Saito, and M. Shapiro, “Optimistic Replication,” *ACM Computing Surveys* 37(1):42-81, 3-2005.
[30] S. Schleimer, D. Wilkerson, and A. Aiken, “Winnowing: local algorithms for document fingerprinting,” *Proceedings of the 2003 ACM SIGMOD international conference on Management of data,* pp.76-85, 2003.
[31] T. Suel, P. Noel, and D. Trendafilov, “Improved File Synchronization Techniques for Maintaining Large Replicated Collections over Slow Networks,” *IEEE International Conference on Data Engineering*, 3-2004.
[32] D. Trendafilov, N. Memon, T. Suel, “zdelta: An Efficient Delta Compression Tool,” *Technical Report TR-CIS-2002-02*, Polytechnic University, June 2002.
[33] A. Tridgell, and P. Mackerras, “The rsync algorithm,” *Technical Report TR-CS-96-05*, Australian National University, June 1996.
[34] A. Tridgell, “Efficient Algorithms for Sorting and Synchronization,” *PhD thesis, Australian National University*, 1999.