In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI '07)*, Cambridge, Massachusetts, April 2007

Exploiting Similarity for Multi-Source Downloads Using File Handprints

Himabindu Pucha, David G. Andersen, Michael Kaminsky
Purdue University, Carnegie Mellon University, Intel Research Pittsburgh

Abstract

Many contemporary approaches for speeding up large file transfers attempt to download chunks of a data object from multiple sources. Systems such as BitTorrent quickly locate sources that have an exact copy of the desired object, but they are unable to use sources that serve similar but non-identical objects. Other systems automatically exploit cross-file similarity by identifying sources for each chunk of the object. These systems, however, require a number of lookups proportional to the number of chunks in the object and a mapping for each unique chunk in every identical and similar object to its corresponding sources. Thus, the lookups and mappings in such a system can be quite large, limiting its scalability.

This paper presents a hybrid system that provides the best of both approaches, locating identical *and* similar sources for data objects using a constant number of lookups and inserting a constant number of mappings per object. We first demonstrate through extensive data analysis that similarity does exist among objects of popular file types, and that making use of it can sometimes substantially improve download times. Next, we describe *handprinting*, a technique that allows clients to locate similar sources using a constant number of lookups and mappings. Finally, we describe the design, implementation and evaluation of Similarity-Enhanced Transfer (SET), a system that uses this technique to download objects. Our experimental evaluation shows that by using sources of similar objects, SET is able to significantly out-perform an equivalently configured BitTorrent.

1 Introduction

Extensive research over the past several decades has explored many techniques to improve data transfer speed and efficiency. Despite this effort, bulk data transfers often remain slow for a variety of reasons. First, of course, receivers may be bandwidth-limited. Second, the source or sources may be unable to saturate the receiver's bandwidth. Third, congestion or failures in the "middle" of the network may slow the transfer.

Downloading data from multiple sources simultaneously is a popular technique to speed transfers when the receiver is not the bottleneck. Many peer-to-peer content distribution systems use this idea, for example. These systems employ two basic strategies to locate the sources of data: per-file and per-chunk.

In a per-file system, receivers locate other sources of the *exact* file they are downloading in $O(1)$ lookups. These systems, exemplified by BitTorrent [7], Gnutella [1] and ChunkCast [6], typically use a global location service. Unfortunately, as we explore further in Section 2, the performance of file transfers using these systems is often unacceptably slow, with users requiring hours or even days to download content.

In a per-chunk system, receivers locate sources for individual pieces, or chunks, of the desired file. Since any given chunk in a file might appear in several other files, finding sources for each chunk can yield a much larger number of *similar* sources. The cost, however, is performing $O(N)$ lookups, one for each of the N chunks in the file the receiver is trying to download. Moreover, such systems also require a mapping between every unique chunk in the identical and similar files and their corresponding sources, i.e., $O(N)$ mappings per object. Examples of per-chunk systems are CFS [9] and Shark [3].

In this paper, we present Similarity-Enhanced Transfer (SET)—a hybrid system that provides the best of both approaches, locating both identical *and* similar sources for data chunks using $O(1)$ lookups and by inserting $O(1)$ mappings per file. We justify this approach by demonstrating that (a) cross-file similarity exists in real Internet workloads (i.e., files that people are actually downloading on today's file-sharing networks); (b) we can find similar sources in $O(1)$ lookups; (c) the extra overhead of locating these sources does not out-weigh the benefit of using them to help saturate the recipient's available bandwidth. Indeed, exploiting similar sources can significantly improve download time.

The three contributions of this paper are centered around these points. First, we present a detailed similarity analysis of 1.7 TB of data fetched from several active file-sharing networks. These data represent a variety of file types, with an emphasis on multimedia files often ignored by previous similarity studies. Our results show that significant cross-file similarity exists in the files that are frequently transferred on these networks. By exploiting this similarity, receivers can locate several times the number of potential sources (Section 3).

The second contribution of this paper is a technique to locate similar sources to the file being downloaded using only $O(1)$ lookups. This technique, which we term *handprinting*, is a novel use of deterministic sampling. Sources insert a fixed number of hashes into a global database; receivers look up their own set of hashes in this database to find sources of similar files. System-wide parameters determine the amount of similarity receivers can detect (e.g., all files with $x\%$ similarity to the target file) and with what probability they can detect it (Section 4).

Third, to demonstrate the benefit of this approach to multi-source downloads, we built a prototype system that uses handprinting to locate sources of similar files (Section 5). Our results show that the overhead of our approach is acceptable (roughly 0.5% per similar file). Without using similar sources, the prototype meets or exceeds BitTorrent's performance. When we enable downloads from similar sources, the system finds and uses these sources to greatly improve transfer speeds (Section 6).

2 Motivation: Slow Downloads

With fast, asymmetric broadband connections, receivers are frequently unable to saturate their available bandwidth during downloads, even when using existing multi-source peer-to-peer download programs. For example, in 2003, Gummadi et al. found that 66.2% of downloads failed, and that the median transfer time in Kazaa for files over 100 MB was *over one day* [13]. In 2002, 75% of Gnutella peers had under 1 megabit of upstream capacity [28].

Unfortunately, our experience suggests that this situation has not improved in the intervening years. Figure 1 shows the CDF of throughput achieved while downloading 6208 large files from popular file-sharing networks (the details of our measurement study appear in Section 3.1). The median transfer achieved under 10 Kbit/s of average throughput, and the 90th percentile only managed 50 Kbit/s, despite running these experiments from an extremely well-connected academic network.

While they are a step in the right direction, today's file-sharing applications clearly have room for improvement. We hypothesize that this improvement can come from giving these applications additional sources from which to download—specifically, by allowing them to download from sources with files similar to their target file.

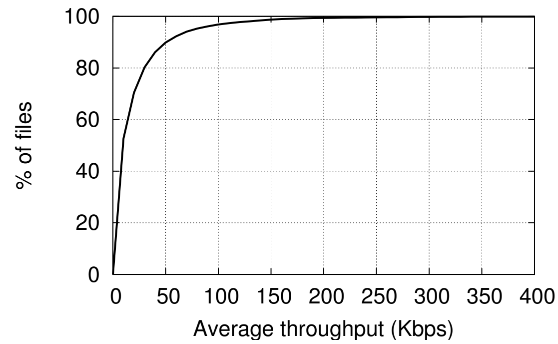

Figure 1: Throughput observed in 6208 file downloads from file-sharing networks.

3 Similarity

We care about *exploitable* similarity that can be used to speed transfers. In most multi-source systems, including the one we propose here, the smallest unit of data transfer is a chunk. Chunks can have a static or variable size. To be able to detect similarity between mis-aligned objects, we define the chunk boundaries using Rabin fingerprinting.[1] Exploitable similarity means that another file shares entire chunks with the target file that the receiver is trying to download. We therefore define similarity as the fraction of chunks that are shared between two files.[2]

Prior studies have found exploitable similarity in filesystems (particularly source code and compiler output) [19], email attachments [31, 10], HTML documents and web objects [18, 11], and operating system and application software [8]. The amount of similarity ranged from a few percent for compressed files in filesystems, to 15% for Web objects, to 30% for email.

We were concerned, however, that such similarity would not arise in heavily-compressed multimedia files, which comprise the bulk of traffic on file-sharing networks. To our surprise, we found considerable similarity in these files. This similarity arose for many reasons; examples include:

- MP3 music files with identical sound content but different header bytes (artist and title metadata or headers from encoding programs) were 99% similar.
- Movies and trailers in different languages were often 15% or more similar.

[1] Rabin fingerprinting, as used in systems such as LBFS [19] determines chunk boundaries by examining the hash of a sliding window over the file data, and declaring a chunk boundary when the N lowest bits of the hash are equal to some fixed value (e.g., zero). Using this mechanism, small changes to the content *within* a chunk, including adding or removing bytes, will generally not change the boundaries themselves.

[2] For reasons we discuss further in Section 4, if the files are of different size, we define similarity as the smaller of the pairwise similarities.

- Media files with apparent transmission or storage errors differed in a single byte or small string of bytes in the middle of the file.
- Identical content packaged for download in different ways (e.g., a torrent with and without a README file) were almost identical.

Note that in all of these cases, the modified files (and their hashes) differ from the originals; traditional, per-file systems that look only for identical files cannot find and use the modified versions as additional sources for the original (desired) object. Furthermore, note that our similarity study measures similarity among files exactly as they are offered online for download (e.g., we do not uncompress, recode, etc. media files).

In the rest of this section, we explore the feasibility of using the exploitable similarity among files to improve the performance of multi-source transfers. We propose two metrics to capture the benefit of using similar files. We apply these metrics to a large volume of data downloaded from file-sharing networks to show the effective increase in the number of sources that a client of one of these networks might experience.

3.1 Data Collection

We examined several large software collections as well as 1.7 TB of files downloaded from popular file-sharing networks. The results from analyzing the software archives were similar to those in previous studies. Due to space constraints, we chose not to include them here, but to instead focus on the more novel study of multimedia files.

File sources. We collected files from the eDonkey and Overnet networks using the MLDonkey client [2] for three months between November 2006 and February 2007. We modified the client to record the IP addresses of the sources for files and to not upload blocks.[3]

Selecting files to download. We could only download a fraction of the files available on these file-sharing networks, so the choice of what to download was important. We intentionally sampled a subset of files that were likely to be similar to each other and that we expected people to be interested in downloading.

To do so, we selected groups of unique files returned as search results for a given query. Our hypothesis is that files in these groups are more likely to be similar than would two files chosen completely at random. This approach necessarily provides only a lower bound on similarity that really exists in the file-sharing network (sampling more files would reveal more similarity). To find files that people were interested in downloading, we categorize our search terms as "popular" or "unpopular" (tail). We chose popular terms from recent Billboard top ten audio tracks and top ten box office results. We chose unpopular audio search terms from top song collections between 1990 and 2000, and video terms from one of the authors' personal Netflix history list. We selected terms that resulted in fewer than 50 (but at least one) search result. Our final search terms consisted of 15 popular and 11 unpopular song title/artist pairs plus 14 popular and 12 unpopular movie titles. We ordered the search results by decreasing number of sources and downloaded as many files as possible.

[3]BitTorrent's tit-for-tat mechanism meant that we could not effectively download files using it without also being a source for the files we downloaded.

Cleaning the data. 2567 of the files (78% of them audio files) we downloaded contained garbage. A common source of garbage is services such as Overpeer, which disseminate fake files to discourage sharing copyrighted content [16]. To eliminate obviously incorrect results, we filtered downloads by first compressing with rzip. If the "media" files compressed more than 20%, we eliminated them. Second, we passed MP3s through an MP3-to-WAV converter; if the converter exited with an error, we eliminated the file. Spot checks of the eliminated and retained files suggested that this method produced few false positives and negatives. These garbage files are excluded from all results in this paper—we analyzed only the remaining 6208 files.

3.2 Summary of Similarity Study

Table 1 summarizes the data we collected, broken down by the top three file types. MP3 files had a high median similarity: if one MP3 file was similar to another, it was likely to be nearly identical. In contrast, video files generally had lower similarity. One reason we found for such similarity was multiple versions of the same video in different languages. Finally, we note that the median number of similar files is lower for video files due, in part, to the dataset having fewer such files to examine.

3.3 Quantifying the Benefits of Similarity

The basic question we would like to answer is how much does exploiting the observed similarity among files speed up a data transfer. We address this question in two stages. The first step, described in this section, is analytical. We define two *parallelism metrics* and apply them to the data we downloaded. The metrics quantify, on real data, the similarity in the data and the resulting speedup in a transfer from using that similarity. The second step, described in the following sections, is to understand the practicality of using similar sources through the design, implementation, and evaluation of a system that finds and uses such sources for faster multi-source downloads.

File Type	Count	Median Size (MB)	Median Avg. Throughput (Kbps)	Median Max. Throughput (Kbps)	Median Identical Sources	Median Similarity (%)	Median # Similar Files
MP3	3725	5.3	5.4	31.2	1.1	99.6	25
AVI	1624	699.7	28.8	220.0	6.7	22.9	3
MPEG	533	692.3	22.5	169.6	5.1	99.0	1

Table 1: Similarity in different types of media. Max. throughput refers to the maximum throughput observed during the course of a file download. Median similarity is the median across files with non-zero similarity; the last column shows the median number of files that were similar to each file of this type.

The parallelism metrics attempt to quantify how many different sources a receiver can draw from to download a particular object. One metric assumes limited parallelism ("conservative") and the other assumes perfect parallelism ("optimistic"). This analysis assumes that all sources serve chunks at the same rate and that chunks are the same size.

Both metrics produce similar results when files are either extremely similar or extremely dissimilar. They capture the decreasing benefit of having many sources for only a small subset of the chunks (because the few sources for the remaining chunks will be the bottleneck):

- Two sources that each have almost all of the chunks of the file provide a parallelism approaching 2. Three such sources provide a parallelism of nearly 3, and so on.

- If there are many sources for one chunk in the file, but only one source for all of the remaining chunks, the parallelism is only negligibly over 1.

The two metrics differ in how they treat sources with moderate similarity. Imagine that one source has all of the blocks of the file and another source has half of the blocks. The optimistic metric assumes that a receiver could obtain half the blocks from one source and half from the other, producing a parallelism of two. The conservative metric instead accounts for the loss of flexibility in this scenario compared to two identical sources, and assigns it a value of $1\frac{1}{3}$, for reasons we discuss below.

The optimistic metric assumes that the client can download from all sources in parallel with an optimal choice of which chunk to download from which source. Under this optimal assignment of chunks to sources, each source serves C_s chunks out of the total number of chunks C. The time to download the file, therefore, is determined only by the maximum C_s, the largest number of chunks the receiver must download from any single source. The parallelism, then, is $\frac{C}{max\ C_s}$.

The conservative parallelism metric states that a chunk can be downloaded in time inversely proportional to the number of sources for that chunk.[4] Let:

$$\begin{aligned} C &= \text{The number of chunks in the file} \\ S_i &= \text{The number of sources for chunk } i \\ \frac{1}{S_i} &= \text{The time to download chunk } i \end{aligned}$$

Conservative parallelism is then the ratio of the original download time C (one unit of time per chunk) over the parallel download time:

$$\text{Parallelism} = \frac{C}{\sum_{i=1}^{C} \frac{1}{S_i}}$$

The conservative metric assumes *limited* parallelism. Chunks are downloaded one-at-a-time, but any given chunk is downloaded from all available sources in parallel. In the example above with one complete source and one source with half of the chunks, the chunk download time would be $\frac{C}{\frac{C}{2}\cdot\frac{1}{2}+\frac{C}{2}\cdot\frac{1}{1}} = \frac{4}{3}$. If instead these two sources each had a complete copy, the parallelism would be $C/\left(C\cdot\frac{1}{2}\right) = 2$.

Which metric is correct depends on the capacities of the sources and the network topology. Shared bottlenecks and differing source capacities will make the conservative metric more accurate; independent sources with similar capacities, plus intelligent assignment of chunks to sources, will favor the parallel metric.

The analysis that follows uses both metrics to provide a better picture of the potential benefits. We examine *parallelism gain*, the ratio of parallelism with similar sources to parallelism without similar sources.

3.3.1 Analysis Method

To compute the available parallelism, every 10 minutes during a transfer, we logged both the active IP addresses from which our clients actually downloaded a file as well as the set of addresses returned during a search for the file. We combined the search results by IP address: if a host serves F files that could be useful for a download,

[4] In our system the smallest granularity for download is a chunk, but in aggregate parallelizing within a chunk is equivalent to parallelizing between chunks that are available from the same number of sources.

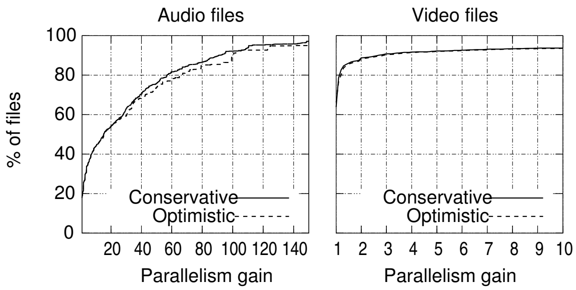

Figure 2: Optimistic and conservative parallelism gain with 16 KB chunk size. Note that the x-axes differ.

the parallelism metrics treat it as single source serving the union of useful chunks across all F files.

We define the set of sources for a file by choosing the active IPs from one random interval during the file transfer. (The results were the same using other choices for the sources: active IPs vs. search IPs, and one random interval vs. the union of all IPs observed during the transfer. The choice did not matter because parallelism gain is *relative*: an increase from 2 identical sources to 4 identical plus similar sources is equivalent to an increase from 20 to 40.)

3.3.2 Optimistic vs. Conservative Parallelism

Figure 2 shows the CDF of parallelism gain for audio and video files split into 16 KB chunks with Rabin fingerprinting. This graph shows three important points. First, 20% of the audio files and 65% of video files were completely dissimilar to the other files we downloaded (they had no exploitable similarity). Second, as expected, the parallelism is higher for the dataset with more samples (audio). With a larger sample size, the study would have discovered a larger number of similar sources for the video files, but (unless our sampling was in some way biased), it would not have discovered a greater degree of similarity among those sources.

3.3.3 Chunk Size and Parallelism

Figure 3 shows the CDF of parallelism gain with chunk sizes that vary from 2 KB to 128 KB. Smaller chunk sizes take advantage of finer-grained similarity between files, but larger chunk sizes impose lower overhead. We set our chunk size to 16 KB, which imposes less than 1% per-chunk overhead in our implementation for identifying and requesting the chunks, but gains most of the parallelism.

3.3.4 Popularity and Parallelism

Figure 4 shows that there is slightly more parallelism gain for popular files than for unpopular files. We attribute this difference primarily to finding a smaller number of "variants" of the less popular content. The parallelism gain for both categories, however, is still significant and can be successfully exploited for objects in both categories.

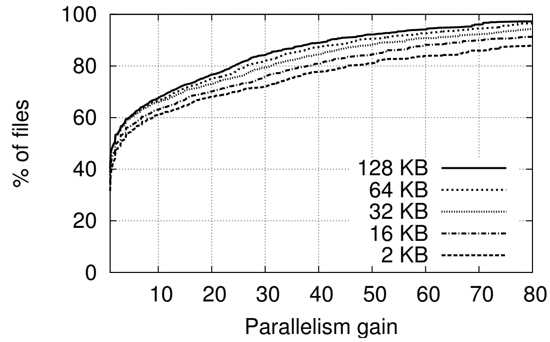

Figure 3: Conservative parallelism gain as chunk size is varied for media files using Rabin fingerprinting.

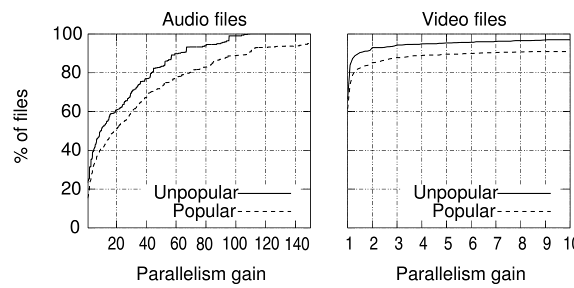

Figure 4: Conservative parallelism gain for files of different popularity.

In summary, many files available on popular file-sharing networks have exploitable similarity that could be used to increase substantially the parallelism available for transfers. The majority of transfers are slow (Figure 1), and these slow transfers have available significant parallelism when using similar files. We therefore believe that allowing clients to use these similar sources could significantly reduce their download time, if only they had a way to do so.

4 Handprinting to Find Similarity

This section presents handprinting, our technique for finding useful sources of shared content. Handprinting is inspired by earlier techniques ("shingling," "fingerprinting," and deterministic sampling) often used to detect similarity between files (e.g., for clustering search results [4] or detecting spam [33]). Our contribution is the adaptation and analysis of these techniques to efficiently locate sources of *exploitable* similarity during file transfer; prior work used them purely as a similarity *metric*.

Handprinting first divides each file into a series of chunks, $C_1 \cdots C_N$, obtained through Rabin fingerprinting.[5] Next, it computes the hash (fingerprint) of each chunk, and takes a deterministic sample of these fingerprints. To sample, handprinting sorts the hashes in lexicographic order, selects the first k hashes (the handprint), and inserts them into a global lookup table. To find similar files, a receiver obtains the chunk hashes for its desired file and searches for entries for any of the first k hashes. The algorithm detects similar files if they collide on any of the k chunks. This process is illustrated in Figure 5, and we analyze the probability of its success below.

4.1 Quantifying Discovery Probability

Consider two files A and B, composed of N_A and N_B chunks, respectively. A and B have m distinct chunks in common (the shaded chunks in Figure 5).

We assume that the hash of a chunk is deterministic (the hash of the same data produces the same value), but that the actual value is effectively random. What is the probability that handprinting detects two shared files with particular values of N_A, N_B, m, k?

Because the chunk hashes are sorted, the handprint will contain shared chunks from the two files *in the same order*. Thus, if the handprints of both files A and B contain at least one shared chunk hash, that chunk hash is guaranteed to be the same chunk in both files A and B. This is unlike randomly selecting k chunk hashes, in which even if the handprints of both files contain a shared chunk, it may not be the same chunk, which thus has a lower probability of intersection. The probability of detection p, then, is simply the probability that the handprint of each file includes one or more shared chunks:

$$\begin{aligned} p &= P(\geq 1 \text{ shared chunk from A}) \cdot \\ &\quad P(\geq 1 \text{ shared chunk from B}) \end{aligned}$$

Assuming that the file is large enough to sample with replacement,[6] the probability of choosing at least one shared chunk from a file is:

$$\begin{aligned} P(\text{pick shared}) &= 1 - P(\text{all } k \text{ chunks not shared}) \\ &= 1 - (P(\text{one chunk not shared}))^k \\ &\geq 1 - \left(1 - \frac{m}{N}\right)^k \end{aligned}$$

Thus, the total probability of detecting that the files share chunks is:

$$p \geq \left(1 - \left(1 - \frac{m}{N_A}\right)^k\right) \cdot \left(1 - \left(1 - \frac{m}{N_B}\right)^k\right)$$

4.1.1 Defining similarity

If the files are the same size, the probability of detection depends only on k and on the *fraction* of chunks in the files that are shared ($\frac{m}{N}$). This fraction is the *similarity* of the two files, denoted s.

Similarity is more complicated when the two files are of different size. If they share m chunks, is the similarity $\frac{m}{N_A}$, $\frac{m}{N_B}$, or a combination? To simplify analysis, we define the similarity as the minimum of these two similarities. As a result, the probability we established for detecting similarity becomes a slightly more pessimistic lower bound, but the inequality remains correct.

This definition is not merely a simplification, however, since it presents an important limitation of handprinting: Given a target file, it is easier to find shared chunks in files *smaller* than the target than in files larger than the target. With the same number of shared chunks, smaller files will have a larger similarity value than larger files. In practice, however, most of the cases of file similarity that we have observed occur between files of similar size.

4.1.2 How many chunk samples?

The value of k should balance a high chance of detecting similar files with the overhead of performing queries. We can select k to have a certain probability of detection p for files with similarity s:

$$p \geq \left(1 - (1 - s)^k\right)^2$$

solving for k:

$$k \geq \frac{\log\left(1 - \sqrt{p}\right)}{\log(1 - s)}$$

Thus, for example, $k = 28$ provides at least a 90% chance ($p \geq 0.9$) of finding a file that shares 10% or more of its chunks ($s \geq 0.1$) with the target file. Table 2 shows the values of k required to detect 90% of files with a varying similarity threshold. Handprinting with $k = 30$ successfully found over 99% of files in our data set with over 10% similarity. (Recall that for files in our data set that shared chunks, the median similarity was higher than 10%. The probability of finding such files was therefore higher than our 90% target.)

4.1.3 Handprints versus random chunk selection

Handprinting substantially improves the probability of finding similar files versus a naive random chunk selection mechanism. In the limit, consider a scheme that selects one chunk at random from each file. The probability that this scheme will work is the probability of selecting a shared chunk from the first file ($\frac{m}{N}$) and also selecting the

[5] We use Rabin fingerprinting only to determine the chunk boundaries, not to fingerprint the chunks.

[6] A pessimistic assumption; sampling without replacement increases the chances of having at least one collision, but the assumption is reasonable for the large file transfers that handprinting targets.

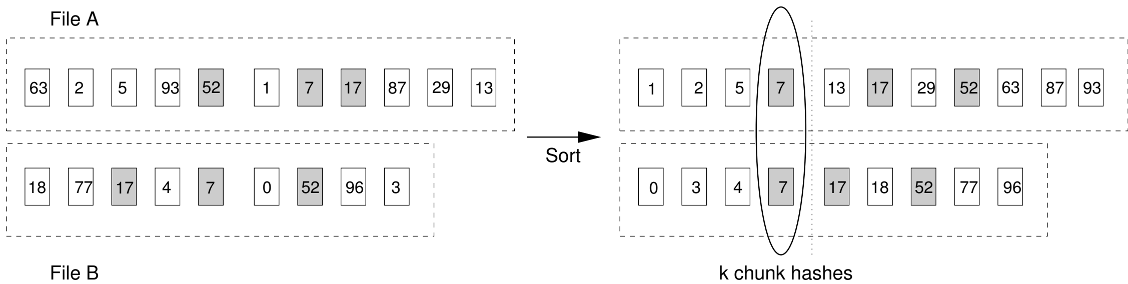

Figure 5: Handprinting. By sorting the chunk hashes before insertion, handprinting imposes a consistent order on hashes so that if a set of one or more shared (shaded) hashes are chosen from each file, the sets will intersect.

Target Similarity (s)	k
90%	1.29
50%	4.29
10%	28.19
5%	57.90
1%	295.55

Table 2: Number of chunk samples for p = 0.9 as similarity is varied.

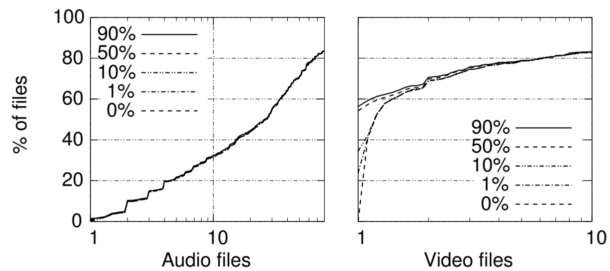

Figure 6: Conservative parallelism as the similarity threshold is varied for smaller audio files and video files. Note that the x-axis of the two graphs differs. The five lines on the left graph completely overlap.

same chunk from the second file ($\frac{1}{N}$), or $\frac{m}{N^2}$. If the files share 10 out of 100 chunks, the probability of detecting them using the random scheme is 0.001; using handprints, the probability is 0.01, an order of magnitude better.

4.2 Similarity Threshold and Parallelism

Handprinting finds files with a particular threshold similarity. How does this affect the parallelism from the previous section? The answer depends on the distribution of similarity in the files. Figure 6 shows the parallelism with different thresholds, from using all files to using only those that are 90% similar. The left graph shows the curve for audio files, which have a high median similarity. As a result, the available parallelism for these files does not depend greatly on the threshold. The right graph shows the parallelism for video files, which rely more on being able to find less-similar files. We choose a 10% similarity threshold as a generally applicable value that does not impose excess overhead.

5 Design and Implementation

This section describes three aspects of the design of SET. First, we examine how the system determines handprints, how it inserts them into a global lookup table, and how receivers query this table to discover similar sources. Next, we describe the surrounding multi-source framework that we built to download files using handprinting. Finally, we briefly discuss the implementation of SET.

5.1 Locating sources with handprinting

SET allows receivers to download objects from multiple sources, which contain either exact or similar objects, using a constant number of lookups. Every object in SET is identified by its Object ID (OID), a collision-resistant hash of the object. To download a file, the receiver must locate the sources for the actual file it wishes to download (*exact sources*) and for the other files that share chunks with the target file (*similar sources*). We assume the receiver already has a complete list of chunk hashes for the desired object. (For example, such hashes are included as part of the BitTorrent "torrent" file.) The receiver uses these hashes to fetch the individual chunks from the sources it discovers.

For every object that a source makes available, it inserts a set of mappings into a global lookup table. This lookup table could be a Distributed Hash Table (DHT), or a service run by a single organization such as Napster. The lookup table contains two types of mappings: (1) chunk hash → OID, and (2) OID → source.

Mapping 1 associates the hash of a particular chunk in the object with the object identifier for that object. There are exactly k mappings that point to each OID, as defined in Section 4. These mappings allow receivers to locate objects that are similar to the one they want to download.

Mapping 2 associates the OID of an object with the

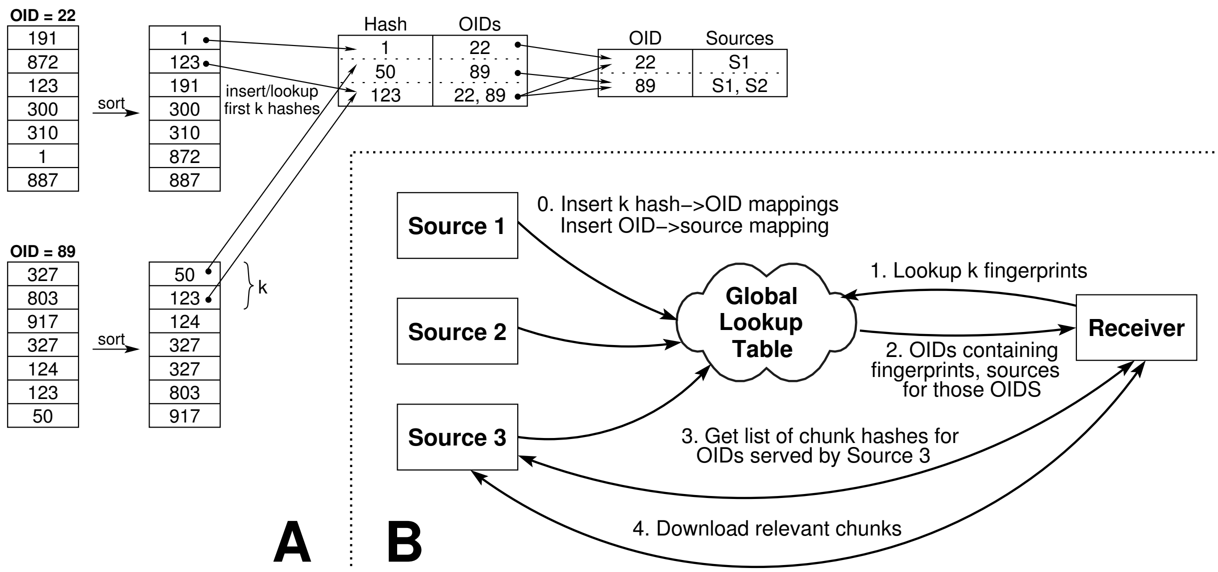

Figure 7: A—Mapping in SET. Mapping 1 links chunk hashes to OIDs; mapping 2 links OIDs to sources of those OIDs. B—SET design overview.

sources for that object. The lookup table contains one such mapping for each source that serves the corresponding object. If three sources serve OID x, then three OID-to-source mappings will exist in the lookup table (but still only k chunk to OID mappings, since the selected chunks will always be the same for a given OID). The mappings are shown in Figure 7-A.

Given the list of chunk hashes, the receiver computes the object's handprint (the k chunk hashes), and looks each hash up in the global lookup table. The result of this query is a list of OIDs that are exploitably similar to the desired file (they share one or more large chunks of data). For each OID, the receiver queries the lookup table to determine the list of potential sources. Finally, the receiver must query one source for each OID to obtain the list of hashes for that OID. At this point, the receiver knows a set of sources that have or are in the process of downloading each of the chunks in the object the receiver wishes to download. This process is illustrated in Figure 7-B.

This process requires a constant number of lookups regardless of the file size: k queries to the distributed lookup table to determine the similar files, plus $2\times$ the number of similar OIDs (first to find potential sources for each OID and second to determine the list of hashes for each similar OID). SET limits the number of similar OIDs to a constant (in our implementation, 30), preferring OIDs that matched more chunk hashes if it has an excess number of choices.

Given the list of potential sources for each chunk in the desired file, the receiver can begin downloading the chunks in parallel from the sources.

5.2 Retrieving Chunks

The literature is rich with optimizations for faster ways to download chunks from multiple sources in "swarming" or "mesh" transfers [7, 14, 5, 29, 20]. Our focus in this paper is on evaluating the benefits from handprinting, which we believe apply to many of these systems. We therefore implement a set of capabilities similar to those in BitTorrent for deciding which sources to use for which chunks, though we plan in the future to incorporate more advanced optimizations into SET.

Maintaining an up-to-date list of sources. New sources appear for chunks in two ways:

1. A new network source arrives
2. An already found source obtains a new chunk.

SET periodically queries the global lookup table for new sources for the OIDs it has discovered, and for new OIDs that are similar to its target file. This low-rate periodic refresh is particularly important when downloading large files, which may take sufficient time that many sources may arrive or depart during the download.

On a finer timescale, SET directly queries other clients downloading exact and similar files to retrieve a bitmap of which chunks the clients have. Because they may not have received the entire file, we term these other clients *partial* sources. These partial sources insert mappings into the global table just like complete sources do.[7]

[7] BitTorrent uses asynchronous notifications that a new hash has arrived. While the overhead from querying the entire bitmap is not a bottleneck in our evaluation, we plan to adopt asynchronous notifications to reduce overhead when transferring extremely large files.

Source selection heuristics. SET uses the *rarest-random* strategy [14], selecting chunks uniformly at random from the rarest chunks. It places chunks in a list with other chunks available from the same number of sources, and processes these lists in order. SET also implements an "endgame mode" [15] similar to BitTorrent's. To avoid waiting excessively for slow sources at the end of the transfer, SET requests all outstanding but not yet complete chunks from an additional randomly selected source.

5.3 Implementation

SET is implemented as a set of modules within the Data-Oriented Transfer modular framework [31].

SET implementation. DOT determines an OID as the SHA-1 hash of a file, and a *descriptor list* as the set of chunk hashes for a file. Internally, SET is added to DOT as two new modules. The first is a *storage plugin* that sits between DOT and the disk. When the client receives an object, the SET storage plugin computes the handprint for the object using the descriptor list handed to it from DOT and inserts it into the distributed lookup table. This insertion allows other nodes to locate the data objects cached by a particular node.

The second module is a *transfer plugin*. DOT uses transfer plugins to send and receive chunks across the network. Transfer plugins implement a simple API:

```
get_descriptors(oid, hints)
get_chunk(descriptor, hints)
```

Hints tell the transfer plugin which sources it should attempt to contact to retrieve the descriptor list or the data chunk. The SET transfer plugin sits between DOT and its usual RPC-based transfer plugin. When DOT first requests an object from the SET transfer plugin, the plugin examines the descriptor list returned from the sender, computes the object's handprint, and queries the global lookup table to find similar sources. The plugin then assigns sources to each chunk by setting the chunk's hints according to the source selection heuristics described above. The multi-source transfer plugin then uses these hints to download the actual chunks.

Changes to DOT. SET is one of the first complex transfer plugins developed under DOT, and it exposed two limitations of DOT's original interface. To correct these, we first added to the transfer plugin API a `notify_descriptors(oid, descriptors)` method that lets transfer plugins associate the descriptors they are fetching with the OID they belong to. In the original DOT API, transfer plugins would have only received a chunk request, with no way to find sources for the chunk from similar files.

Next, we added a local and remote `get_bitmap` call to determine which chunks of an object were available at a node. While a DOT plugin could determine this information locally by querying for each descriptor separately, we felt that the efficiency gain justified adding a specialized interface.

For efficiency, we also changed DOT's representation of OIDs and descriptors from 40 byte hex-encoded ASCII strings to 20 byte binary objects. While the original design favored ease of debugging and tracing with a human-readable string, SET retrieves several descriptor lists for each object it downloads, increasing the overhead imposed by the prior ASCII representation.

Distributed Lookup Table. SET inserts and looks up map entries through an RPC put/get interface. The interface supports either OpenDHT [26] (a public, shared distributed hash table that runs on PlanetLab), or our simplified, centralized implementation (*cDHT*). Our evaluation uses cDHT to isolate the results from performance variation due to OpenDHT.

6 Evaluation

Our evaluation of SET examines whether our design and implementation is able to effectively discover and use additional sources of similar objects. The evaluation uses a mix of synthetic data and the data collected for the similarity analysis described in Section 3. We evaluate SET using simplified topologies in Emulab [32] to examine different aspects of its performance in isolation, and in several deployments on PlanetLab [23] ranging from 9–50 nodes. To ensure that the benefits achieved by taking advantage of similar sources are not simply compensating for flaws in our implementation, we compare the performance of SET to that of BitTorrent where appropriate. For the comparison, we disabled BitTorrent's tit-for-tat mechanism to allow it to send at full speed to all clients.

The results of our evaluation are promising: SET matches or exceeds BitTorrent's performance on the same topologies without using similarity. Using even modest amounts of similarity substantially improves SET's performance, handily outperforming the stock BitTorrent client.

Despite being pleased with our system's performance, we note that we do not present this comparison to show that our multi-source downloading is *better* than that of BitTorrent or other protocols. Our contributions are the design and evaluation of efficient mechanisms for discovering similar sources, not any particular optimization in conventional per-file peer-to-peer transfers. Under many circumstances, it is likely that research peer-to-peer systems such as Bullet Prime [14], ChunkCast [6], or CoBlitz [22] could outperform our prototype when SET is not using similarity. Rather, we present the comparison to show that SET's performance is in line with that of other modern download clients and that these systems too could benefit from using handprinting.

6.1 Using similar sources significantly improves transfer speeds

The experiments described in this section are run on Emulab with five receivers attempting to download from one origin source with a variable number of similar sources. Each node is connected to a 100 Mbit/sec "core" via an access link whose size varies by the experiment. To isolate the causes of slowdowns, the access links are configured so that the core is never the bottleneck.

The Emulab experiments compare SET to BitTorrent across four different network configurations. In each configuration, all nodes have the same speed access link. The configurations are:

Slow DSL	384 Kbit/s up, 1500 Kbit/s down, 40 ms RTT
Fast DSL	768 Kbit/s up, 3000 Kbit/s down, 40 ms RTT
T1	1500 Kbit/s symmetric, 40 ms RTT
High BDP	(High bandwidth delay product) 9800 Kbit/s symmetric, 400 ms RTT

For each network configuration, we measure download time for four scenarios:

BT	BitTorrent, no similar sources possible
SET	SET, no similar sources
SET10	SET, low similarity (one 10% similar source)
SET15	SET, modest similarity (three 15% similar sources)
SET90	SET, high similarity (three 90% similar sources)

In the first two scenarios, the five downloaders can download only from each other and from the origin. This provides a baseline comparison of SET and BitTorrent. The next three scenarios add increasing numbers of similar sources to see if SET can use them effectively. We examine sources that are 10%, 15%, and 90% similar. 10% is SET's target minimum similarity. 15% and 90% are the median similarity for video and audio files from our data set, respectively.

Figures 8, 9 and 10 show the average, maximum, and minimum download times for clients for 4 MB, 10 MB and 50 MB files in these scenarios over two runs.[8] SET slightly outperforms BitTorrent without using similar sources. By using similar sources, SET can perform up to three times faster. Using only a single 10% similar source reduces the average download time by 8%, showing the benefits that arise from exploiting even small amounts of similarity. Similarly, three 15% similar sources reduce the download time by 30% on average. Thus, using larger numbers of similar sources provides increasing benefits. Similar improvements occur across all of the file sizes we examined.

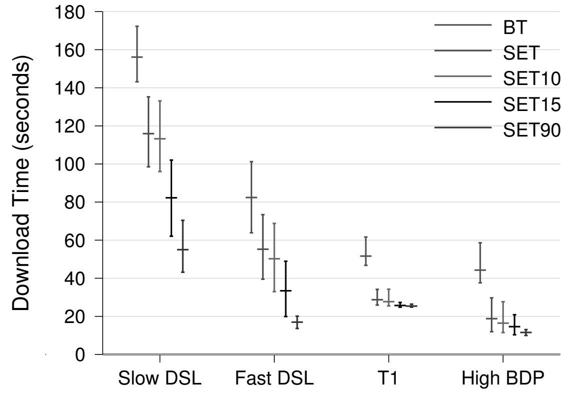

Figure 8: Transfer times for a 4 MB file on Emulab for each transfer method over four network configurations. Within a configuration, the lines appear in the same order as in the legend (BT, SET, SET10, SET15, SET90).

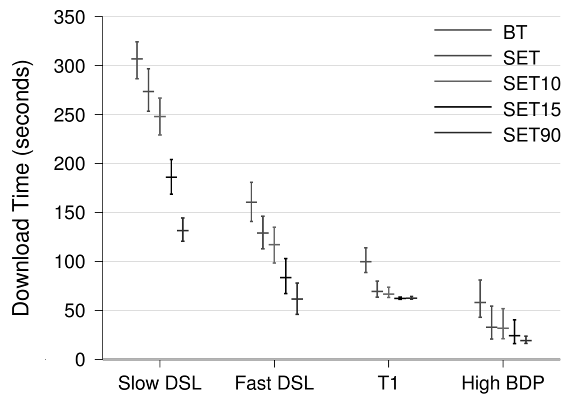

Figure 9: Transfer times for 10 MB file on Emulab

6.2 SET performs well in the real world

We ran the initial experiments (described above) on Emulab in order to isolate the performance effects of varying individual parameters such as file similarity, capacity, and number of sources. In this section, we describe several analogous experiments that we ran on PlanetLab to confirm whether or not SET performs on real networks in the same way it does on an emulated one. These experiments also have five clients downloading from one origin and a variable number of similar sources, except where noted.

[8] We choose 50 MB to facilitate comparability with CoBlitz [20], Shark [3], and Bullet Prime [14].

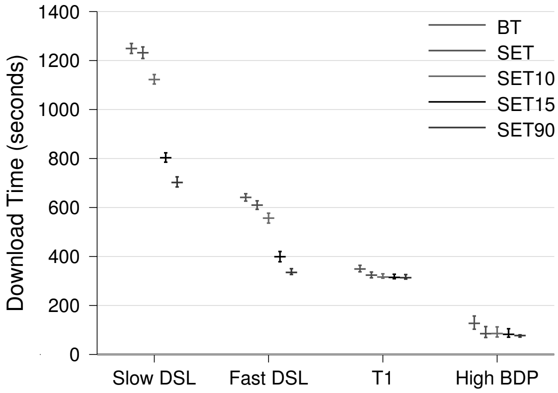

Figure 10: Transfer times for 50 MB file on Emulab

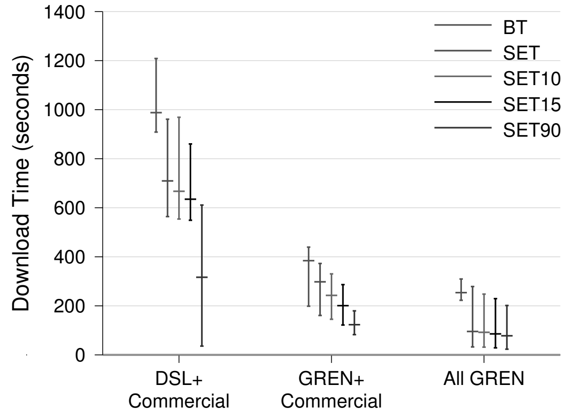

Figure 11: Transfer times for a 50 MB file on PlanetLab for each transfer method over the three configurations.

As in the previous experiments, we find that additional sources of similarity help (and that SET is able to efficiently locate them) to the point that the receiver's available bandwidth becomes the bottleneck. We show this by measuring download time in three scenarios:

- **GREN**: Nine nodes drawn from sites on the Global Research and Education Networks (Internet2, Abilene, etc.). These nodes are extremely well-connected.
- **Mixed**: Nine nodes drawn from commercial and smaller educational networks. This set of nodes is more representative of well-connected commercial sites. There are few enough GREN nodes that most of the transfers occur over commercial Internet paths.
- **DSL+Commercial**: Eight DSL-connected PlanetLab nodes and commercial sites. In this experiment, the similar sources that are added are all moderately well-connected commercial sites, and the clients are mostly DSL connected. This experiment had only four clients because of the limited number of extremely slow clients available.[9]

Figure 11 shows the performance of SET in these PlanetLab scenarios when transferring a 50 MB file. The numbers presented are the average, min, and max of all of the clients across two runs. Several conclusions are apparent from these graphs.

First, when running on the high-bandwidth GREN nodes, the benefits of additional sources are minimal, because senders can already saturate receivers. There is a small reduction in the maximum transfer time from these additional sources, but we suspect this time is no different from that of using multiple TCP connections to a single sender. The larger difference between BT and SET is a combination of 1) SET's ability to immediately begin downloading from the origin before contacting a tracker (or equivalent), and 2) the interpreted BitTorrent client requiring slightly more CPU time under contention on the PlanetLab nodes.

[9]Only three PlanetLab DSL nodes were usable during the times we conducted our experiments.

Second, when running on the more constrained commercial and DSL nodes, using additional sources provides benefit similar to that observed in the Emulab experiments (though with increased variance, as expected).

The huge improvement in transfer times when using three 90% similar sources with the DSL clients (left-hand cluster of lines in Figure 11, rightmost line) arises because one of the similar sources has a faster commercial network connection. When the fast source has only 15% of the blocks, it modestly improves the transfer time, but the overall transfer remains constrained by the slowest 85% of the chunks. However, when the fast source has a 90% similar file, the fast clients can rapidly retrieve most of their chunks from the fast source. While these improvements are in-line with what one would expect, they also show that SET is able to operate effectively when the sources and clients have a diverse range of available bandwidth.

6.3 Scaling with additional similar sources

How well does SET scale as the system has more sources of similar content? In this experiment, the receiver attempts to download a 10 MB file. There is one origin source and a varying number of different 90% similar sources. We examine the same four network configurations used previously (Slow DSL, Fast DSL, T1, and High BDP).

Figure 12 shows two important properties of SET. First, it effectively uses additional, similar sources to speed transfers when the sources are the bottleneck. For the slow

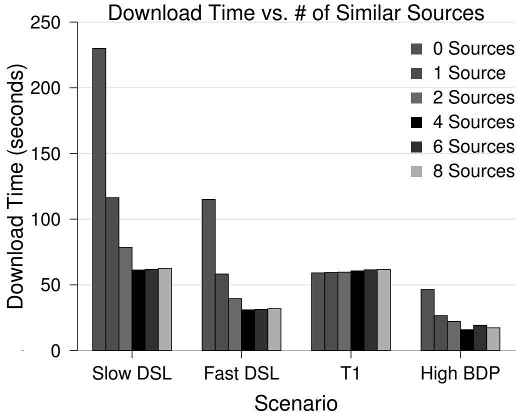

Figure 12: Scaling with a variable number of similar sources and one exact source

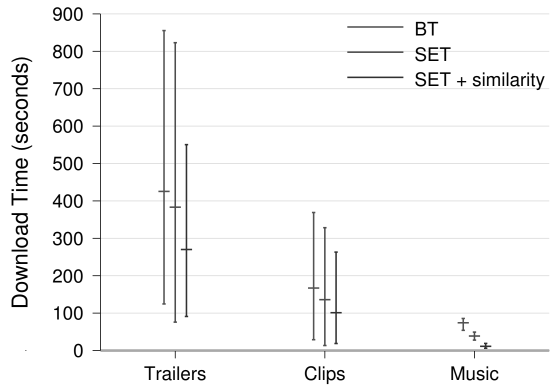

Figure 13: Transfer times for real scenarios on PlanetLab for each transfer method.

DSL scenario, the benefits increase until the receiver's capacity is saturated by $\frac{1500}{384} \approx 4$ sources. The fast DSL behaves similarly, while the symmetric T1 link can be saturated by a single source. As expected, using similar sources also helps when a single TCP connection is window limited and unable to saturate the high bandwidth delay product link.

Second, using similar sources adds minimal overhead in the symmetric case when a single source can saturate the receiver's capacity (the right side of the graph for the T1). The transfer time increases from 59.08 seconds with only the original source to 61.70 seconds with eight (unneeded) similar sources. In general, SET incurs a roughly 0.5% overhead per additional similar file it draws from, because SET must retrieve the full descriptor list for each file, and it must perform periodic bitmap queries from these sources. In practice, we expect the benefits from finding additional sources to outweigh the small overhead required to take advantage of them.

6.4 Scenarios from File-Sharing Networks

We conclude by examining three scenarios drawn from our measurement study of file-sharing networks. For each of these scenarios, we map the sources and clients randomly to a pool of 50 nodes containing all available DSL and commercial nodes on PlanetLab along with randomly chosen GREN nodes.

Different-length movie trailers: Ten receivers attempt to download a 55 MB file from one origin source. They are helped by 21 other sources of a 33 MB file that shares 47% of the target file's blocks.

Minimally-similar movie clips: 14 receivers download a 17 MB file from one origin. They are helped by 32 sources of a 19 MB file that is 15% similar to the target.

Unpopular music: Four receivers download an 11 MB MP3 file from one origin. They are helped by four other clients that have an MP3 file that is 99% similar to the target file.

SET effectively reduces the receivers' download times in these scenarios (Figure 13). Using a 47% similar file reduces the movie trailer download time by 36% and 30% compared to BitTorrent and SET without similarity, respectively. In the second scenario, the 15% similar file reduced the movie clip transfer times by 36% and 26%. Unsurprisingly, the common case of a 99% similar music file was greatly improved using similarity, reducing its transfer time by 71% over SET without similarity.

7 Applicability of SET

From our evaluation, we know that SET works well when there are adequate amounts of exploitable similarity and the original senders cannot saturate the receivers' available bandwidth. Based upon prior work and the analysis in Section 3, we conclude that such similarity exists in a number of important applications: large software updates (e.g., synchronizing a large collection of machines that may start with different versions of the software), transmitting large email attachments, and downloading multimedia content, among others. While our analysis focused on the latter category, we stress that there is nothing in SET that is specific to these file types. We emphasized them in our analysis because of their prevalence on file-sharing networks and because prior studies of similarity did not examine them.

Limited upstream bandwidth appears common in a number of peer-to-peer settings. File transfers in existing networks *are* slow; whether this occurs due to senders explicitly limiting their rates, ISPs policing traffic, or limited upstream speeds does not change the fact that receivers are unable to download at full speed. Regardless of the *cause* of the bottlenecks, being able to draw from more sources increases the likelihood of finding a fast source for the chunks a receiver desires.

8 Related Work

This paper builds on a rich volume of work in several areas: detecting and exploiting similar documents, systems for efficient multi-source bulk file transfer, and peer-to-peer lookup systems.

Similarity detection via "shingling" is a popular approach to creating metrics for the similarity between two or more documents, for purposes of analysis [17], clustering [4] or duplicate elimination [11]. Shingling runs a small sliding window (often 50 bytes) along a document, typically character by character or word-by-word, computing a checksum or hash at each step. Shingling then computes the final fingerprint of a document by deterministically sampling from those hashes. Handprinting differs in goal and analysis more than in mechanism: it is designed to detect exploitable similarity, not document resemblance, and we provide an analysis of its effectiveness when used as the basis for a distributed lookup mechanism. Denehy and Hu also study a similar "sliding-block" technique for filesystem data, finding that more than 32% of the data in their email corpus were duplicates [10].

Several distributed filesystems use chunk-level similarity to reduce duplicate transfers or storage requirements, including the Low Bandwidth File System [19], Casper [30], and the Pastiche [8] and Venti [25] backup systems. Of these systems, we call particular attention to Pastiche, in which nodes try to back up their data to a "backup buddy" that has substantial overlap in their filesystem contents. Pastiche accomplishes this in a local setting by having nodes send a random subset of their hashes to another node to determine similarity. We believe it would be interesting to explore whether handprints could help scale the Pastiche approach to larger clusters of machines. The work in [24] presents an analysis of chunk level similarity in different workloads.

Systems such as CFS [9] and Shark [3] exploit similarity at the chunk level. CFS stores and retrieves blocks from a DHT while Shark stores and retrieves chunk indexes from a DHT. In contrast to SET, they require $O(N)$ lookups—and perhaps more importantly, they require $O(N)$ state in the DHT. Shark improves on earlier DHT-based filesystems by taking advantage of locality for caching and fetching data.

SET builds upon much recent work in peer-to-peer file transfer [5, 7, 12, 14, 20, 21, 29]. SET draws techniques such as its end-game mode and its chunk selection heuristics from BitTorrent [7] and Bullet Prime [14]. A question for ongoing research is identifying which systems can use handprinting to find similar sources, and which cannot. Single-file mesh-based transfer systems such as Bullet Prime and SplitStream [5] are the next step, since they still transfer pieces of the original file. Extending SET to coding-based approaches such as Avalanche [12] seems more difficult, but poses an interesting challenge.

In our evaluation, we focused primarily on taking advantage of similarity when the number of sources were limited, or the available sources all had limited bandwidth. The focus of many prior per-file approaches (such as CoBlitz [20], and Bullet Prime [14]) was on making extremely efficient use of a larger mesh of nodes. Inasmuch as these protocols are used in a scenario where the overlay multicast mesh they create is capable of saturating the receivers' bandwidth, SET's techniques would provide no improvement. If, however, these systems were used in a situation where there was a nearby source of similar data with much better connectivity to the receiver, using similar sources would provide benefits similar to those we showed in Section 6.

SET depends on a global lookup table such as a Distributed Hash Table [26, 27]. SET is generally agnostic to the specifics of the lookup table; we believe it could run equally well on any lookup table that allows multiple values for a key and that allows some form of updates. Some retrieval systems such as ChunkCast [6] use the locality-aware structure of their global lookup tables (in their case, a DHT) to combine lookups with peer selection; we believe this approach is complementary to SET and handprinting, and that integrating the two is a promising avenue of future research.

9 Conclusion

This paper presented SET, a new approach to multi-source file transfers that obtains data chunks from sources of non-identical, but similar, files. We demonstrated that such similarity exists in active file sharing networks, even among popular multimedia content. SET employs a new technique called handprinting to locate these additional sources of exploitable similarity using only a constant number of lookups and a constant number of mappings per file. Evaluation results from a variety of network configurations showed that SET matches BitTorrent's performance without using similar sources, and exceeds BitTorrent's performance when exploiting similarity. SET's overhead is less than 0.5%.

We believe that handprinting strikes an attractive balance for multi-source transfers. It efficiently locates the sources of exploitable similarity that have the most chunks to contribute to a receiver, and it does so using only a small, constant number of lookups. For these reasons, we believe that this technique is an attractive one to use in any multi-source file transfer system.

Acknowledgments

We are particularly grateful to Niraj Tolia for extensive help with DOT and advice with content-addressable storage; to our NSDI shepherd and reviewers; to Nick Feamster, Mike Freedman, Phil Gibbons and Mary Lou Godbe for their helpful feedback. This work was supported in part by NSF CAREER award CNS-0546551.

References

[1] Gnutella. http://gnutella.wego.com, 2000.

[2] MLDonkey. http://mldonkey.sourceforge.net/.

[3] S. Annapureddy, M. J. Freedman, and D. Mazières. Shark: Scaling file servers via cooperative caching. In *Proc. 2nd USENIX NSDI*, Boston, MA, May 2005.

[4] A. Broder, S. Glassman, M. Manasse, and G. Zweig. Syntactic clustering of the web. In *Proceedings of the 6th International WWW Conference*, pages 1157–1166, Santa Clara, CA, Apr. 1997.

[5] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-bandwidth content distribution in cooperative environments. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)*, Lake George, NY, Oct. 2003.

[6] B.-G. Chun, P. Wu, H. Weatherspoon, and J. Kubiatowicz. ChunkCast: An anycast service for large content distribution. In *Proc. 5th International Workshop on Peer-to-Peer Systems (IPTPS)*, Santa Barbara, CA, Feb. 2006.

[7] B. Cohen. Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, USA, June 2003.

[8] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making backup cheap and easy. In *Proc. 5th USENIX OSDI*, Boston, MA, Dec. 2002.

[9] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, Oct. 2001.

[10] T. E. Denehy and W. W. Hsu. Duplicate Management for Reference Data. Research Report RJ10305, IBM, Oct. 2003.

[11] F. Douglis and A. Iyengar. Application-specific delta-encoding via resemblance detection. In *Proceedings of the USENIX Annual Technical Conference*, San Antonio, Texas, June 2003.

[12] C. Gkantsidis and P. R. Rodriguez. Network coding for large scale content distribution. In *Proc. IEEE INFOCOM*, Miami, FL, Mar. 2005.

[13] K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)*, Lake George, NY, Oct. 2003.

[14] D. Kostic, R. Braud, C. Killian, E. Vandekieft, J. W. Anderson, A. C. Snoeren, and A. Vahdat. Maintaining high bandwidth under dynamic network conditions. In *Proc. USENIX Annual Technical Conference*, Anaheim, CA, Apr. 2005.

[15] A. Legout, G. Urvoy-Keller, and P. Michiardi. Rarest first and choke algorithms are enough. In *Proc. ACM SIGCOMM Internet Measurement Conference*, Rio de Janiero, Brazil, Oct. 2006.

[16] J. Liang, R. Kumar, Y. Xi, and K. W. Ross. Pollution in p2p file sharing systems. In *Proc. IEEE INFOCOM*, Miami, FL, Mar. 2005.

[17] U. Manber. Finding similar files in a large file system. In *Proc. Winter USENIX Conference*, pages 1–10, San Francisco, CA, Jan. 1994.

[18] J. C. Mogul, Y. M. Chan, and T. Kelly. Design, implementation, and evaluation of duplicate transfer detection in HTTP. In *Proc. First Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, Mar. 2004.

[19] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, Oct. 2001.

[20] K. Park and V. Pai. Scale and Performance in the CoBlitz Large-File Distribution Service. In *Proc. 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, May 2006.

[21] K. Park and V. Pai. Deploying Large File Transfer on an HTTP Content Distribution Network. In *Proc. Workshop on Real, Large Distributed Systems (WORLDS)*, San Francisco, CA, Dec. 2004.

[22] K. Park, V. Pai, L. Peterson, and Z. Wang. CoDNS: Improving DNS performance and reliability via cooperative lookups. In *Proc. 6th USENIX OSDI*, San Francisco, CA, Dec. 2004.

[23] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the Internet. In *Proc. 1st ACM Workshop on Hot Topics in Networks (Hotnets-I)*, Princeton, NJ, Oct. 2002.

[24] C. Policroniades and I. Pratt. Alternatives for detecting redundancy in storage systems data. In *Proc. USENIX Annual Technical Conference*, Boston, MA, June 2004.

[25] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*, pages 89–101, Monterey, CA, Jan. 2002.

[26] S. C. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A public DHT service and its uses. In *Proc. ACM SIGCOMM*, Philadelphia, PA, Aug. 2005.

[27] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proc. 18th IFIP/ACM International Conference on Distributed Systems Platforms*, Nov. 2001.

[28] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. Multimedia Computing and Networking (MMCN)*, Jan. 2002.

[29] R. Sherwood, R. Braud, and B. Bhattacharjee. Slurpie: A cooperative bulk data transfer protocol. In *Proc. IEEE INFOCOM*, Hong Kong, Mar. 2004.

[30] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, A. Perrig, and T. Bressoud. Opportunistic use of content addressable storage for distributed file systems. In *Proc. USENIX Annual Technical Conference*, pages 127–140, San Antonio, TX, June 2003.

[31] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil. An architecture for Internet data transfer. In *Proc. 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, May 2006.

[32] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. 5th USENIX OSDI*, pages 255–270, Boston, MA, Dec. 2002.

[33] F. Zhou, L. Zhuang, B. Y. Zhao, L. Huang, A. Joseph, and J. Kubiatowicz. Approximate object location and spam filtering on peer-to-peer systems. In *Proc. ACM/IFIP/USENIX International Middleware Conference (Middleware 2003)*, 2003.