

6-2008

Adaptive File Transfers for Diverse Environments

Himabindu Pucha
Carnegie Mellon University

Michael Kaminsky
Intel Research Pittsburgh

David G. Andersen
Carnegie Mellon University, dga@cs.cmu.edu

Michael Kozuch
Intel Research Pittsburgh

Follow this and additional works at: <http://repository.cmu.edu/compsci>

Published In

Proceedings of the 2008 USENIX Annual Technical Conference (USENIX'08), .

This Conference Proceeding is brought to you for free and open access by the School of Computer Science at Research Showcase @ CMU. It has been accepted for inclusion in Computer Science Department by an authorized administrator of Research Showcase @ CMU. For more information, please contact research-showcase@andrew.cmu.edu.

Adaptive File Transfers for Diverse Environments

Himabindu Pucha^{*}, Michael Kaminsky[‡], David G. Andersen^{*}, and Michael A. Kozuch[‡]
^{}Carnegie Mellon University and [‡]Intel Research Pittsburgh*

Abstract

This paper presents *dsync*, a file transfer system that can dynamically adapt to a wide variety of environments. While many transfer systems work well in their specialized context, their performance comes at the cost of generality, and they perform poorly when used elsewhere. In contrast, *dsync* adapts to its environment by intelligently determining which of its available resources is the best to use at any given time. The resources *dsync* can draw from include the sender, the local disk, and network peers. While combining these resources may appear easy, in practice it is difficult because these resources may have widely different performance or contend with each other. In particular, the paper presents a novel mechanism that enables *dsync* to aggressively search the receiver's local disk for useful data without interfering with concurrent network transfers. Our evaluation on several workloads in various network environments shows that *dsync* outperforms existing systems by a factor of 1.4 to 5 in one-to-one and one-to-many transfers.

1 Introduction

File transfer is a nearly universal concern among computer users. Home users download software updates and upload backup images (or delta images), researchers often distribute files or file trees to a number of machines (e.g. conducting experiments on PlanetLab), and enterprise users often distribute software packages to cluster or client machines. Consequently, a number of techniques have been proposed to address file transfer, including simple direct mechanisms such as FTP, “swarming” peer-to-peer systems such as BitTorrent [4], and tools such as *rsync* [22] that attempt to transfer only the small delta needed to re-create a file at the receiver.

Unfortunately, these systems fail to deliver optimal performance due to two related problems. First, the solutions typically focus on one particular resource strategy to the exclusion of others. For example, *rsync*'s delta approach will accelerate transfers in low-bandwidth environments when a previous version of the file exists in the current directory, but not when a useful version exists in a sibling directory or, e.g., `/tmp`. Second, existing solutions typically do not adapt to unexpected environments. As an example, *rsync*, by default, always inspects previous file versions to “accelerate” the transfer—even on fast

networks when such inspections contend with the write portion of the transfer and *degrade* overall performance.

This paper presents *dsync*, a file(tree) transfer tool that overcomes these drawbacks. To address the first problem, *dsync* opportunistically uses all available sources of data: the sender, network peers, and similar data on the receiver's local disk. In particular, *dsync* includes a framework for locating relevant data on the local disk that might accelerate the transfer. This framework includes a pre-computed index of blocks on the disk and is augmented by a set of heuristics for *extensively* searching the local disk when the cache is out-of-date. *dsync* addresses the second problem, the failure of existing file transfer tools to accommodate diverse environments, by constantly monitoring resource usage and adapting when necessary. For example, *dsync* includes mechanisms to throttle the aggressive disk search if either the disk or CPU becomes a bottleneck in accepting data from the network.

Those two principles, opportunistic resource usage and adaptation, enable *dsync* to avoid the limitations of previous approaches. For example, several peer-to-peer systems [6, 1, 4, 11, 19] can efficiently “swarm” data to many peers, but do not take advantage of the local filesystem(s). The Low Bandwidth File System [13] can use *all* similar content on disk, but must maintain an index and can only transfer data from one source. When used in batch mode, *rsync*'s diff file can be sent over a peer-to-peer system [11], but in this case, all hosts to be synchronized must have identical initial states.

dsync manages three main resources: the network, the disk, and CPU. The network (which we assume can provide all of the data) is *dsync*'s primary data source, but *dsync* can choose to spend CPU and disk bandwidth to locate relevant data on the local filesystem. However, *dsync* also needs these CPU resources to process incoming packets and this disk bandwidth to write the file to permanent storage. Therefore, *dsync* adaptively determines at each step of the transfer which of the receiver's local resources can be used without introducing undue contention by monitoring queue *back-pressure*. For example, *dsync* uses queue information from its disk writer and network reader processes to infer disk availability (Section 4). When searching the receiver's disk is viable, *dsync* must continuously evaluate whether to identify ad-

ditional candidate files (by performing directory `stat` operations) or to inspect already identified files (by reading and hashing the file contents). *dsync* prioritizes available disk operations using a novel cost/benefit framework which employs an extensible set of heuristics based on file metadata (Section 5.2). As disk operations are scheduled, *dsync* also submits data chunk transfer requests to the network using the expected arrival time of each chunk (Section 5.3).

Section 6 presents the implementation of *dsync*, and Section 7 provides an extensive evaluation, demonstrating that *dsync* performs well in a wide range of operating environments. *dsync* achieves performance near that of existing tools on the workloads for which they were designed, while drastically out-performing them in scenarios beyond their design parameters. When the network is extremely fast, *dsync* correctly skips any local disk reads, performing the entire transfer from the network. When the network is slow, *dsync* discovers and uses relevant local files quickly, outperforming naive schemes such as breadth-first search.

2 Goals and Assumptions

The primary goal of *dsync* is to correctly and efficiently transfer files of the order of megabytes under a wide range of operating conditions, maintaining high performance under dynamic conditions and freeing users from manually optimizing such tasks. In our current use-model, applications invoke *dsync* on the receiver by providing a description of the file or files to be fetched (typically end-to-end hashes of files or file trees). *dsync* improves transfer performance (a) by exploiting content on the receiver’s local disk that is similar to the files currently being transferred, and (b) by leveraging additional network peers that are currently downloading or are sources of the files being transferred (*identical sources*) or files similar to the files being transferred (*similar sources*).

Typical applications that benefit from similar content on disk include updating application and operating system software [5], distributed filesystems [13], and backups of personal data. In addition to the above workloads, network similarity is applicable in workloads such as email [7], Web pages [8, 12], and media files [19]. Peer-to-peer networks, mirrors, and other receivers engaged in the same task can also provide additional data sources.

The following challenges suggest that achieving this goal is not easy, and motivate our resulting design.

Challenge 1: Correctly use resources with widely varying performance. Receivers can obtain data from many resources, and *dsync* should intelligently schedule these resources to minimize the total transfer time. *dsync* should work well on networks from a few kilobits/sec to gigabits/sec and on disks with different I/O loads (and thus different effective transfer speeds).

Challenge 2: Continuously adapt to changes in resource performance. Statically deciding upon the relative merits of network and disk operations leads to a fragile solution. *dsync* may be used in environments where the available network bandwidth, number of peers, or disk load changes dramatically over the course of a transfer.

Challenge 3: Support receivers with different initial filesystem state. Tools for synchronizing multiple machines should not require that those machines start in the same state. Machines may have been unavailable during prior synchronization attempts, may have never before been synchronized, or may have experienced local modification subsequent to synchronization. Regardless of the cause, a robust synchronization tool should be able to deal with such differences while remaining both effective and efficient.

Challenge 4: Do not require resources to be set up in advance. For example, if a system provides a pre-computed index that tells *dsync* where it can find desired data, *dsync* will use it (in fact, *dsync* builds such an index as it operates). However, *dsync* does not *require* that such an index exists or is up-to-date to operate correctly and effectively.

Assumption 1: At least one correct source. Our system requires at least one sender to transfer data directly; the receiver can *eventually* transfer all of the data (correctly) from the sender.

Assumption 2: Short term collision-resistant hashes. *dsync* ensures correctness by verifying the cryptographic hash of all of the data it draws from any resource. Our design requires that these hashes be collision-resistant for the duration of the transfer, but does not require long-term strength.

Assumption 3: Unused resources are “free”. We assume that the only cost of using a resource is the lost opportunity cost of using it for another purpose. Thus, tactics such as using spare CPU cycles to analyze every file on an idle filesystem are “free.” This assumption is not valid in certain environments, e.g., if saving energy is more important than response time; if the costs of using the network vary with use; or if using a resource degrades it (e.g., limited write cycles on flash memory). While these environments are important, our assumption about free resource use holds in many other important environments. Similarly, *dsync* operates by greedily scheduling at each receiver; we leave the issue of finding a globally optimal solution in a cooperative environment for future work.

Assumption 4: Similarity exists between files. Much prior work [7, 13, 5, 8, 12, 19] has shown that similarity exists and can be used to reduce the amount of data transferred over the network or stored on disk. A major benefit of *rsync* and related systems is their ability to exploit this similarity.

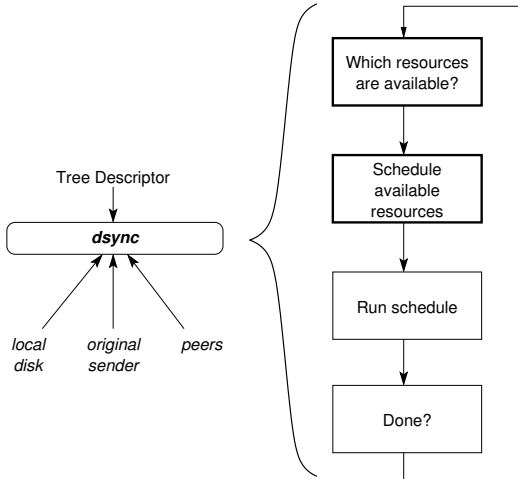


Figure 1: The *dsync* system

3 Design Overview

dsync operates by first providing a description of the file or files to be transferred to the receiver. *dsync* receiver then uses this description to fetch the “recipe” for the file transfer. Like many other systems, *dsync* source divides each file or set of files (file tree) into roughly 16 KB chunks, and it hashes each chunk to compute a chunk ID.¹ Then, *dsync* creates a *tree descriptor* (TD), which describes the layout of files in the file tree, their associated metadata, and the chunks that belong to each file. The TD thus acts as a recipe that tells the receiver how to reconstruct the file tree.

Figure 1 shows a conceptual view of *dsync*’s operation. Given a TD (usually provided by the sender), *dsync* attempts to fetch the chunks described in it from several sources in parallel. In this paper, we focus on two sources: the network and the disk. As we show in Section 7, intelligent use of these resources can be critical to achieving high performance.

3.1 The Resources

dsync can retrieve chunks over the **network**, either directly from the original sender or from peers downloading the same or similar content. Our implementation of *dsync* downloads from peers using a BitTorrent-like peer-to-peer swarming system, SET [19], but *dsync* could easily use other similar systems [4, 1, 11]. Given a set of chunks to retrieve, SET fetches them using a variant of the rarest-chunk-first strategy [10].

dsync can also look for chunks on the receiver’s **local disk**, similar to existing systems such as *rsync* [22]. First, *dsync* consults a pre-computed index if one is available.

¹*dsync* uses Rabin fingerprinting [20] to determine the chunk boundaries so that most of the chunks that a particular file is divided into do not change with small insertions or deletions.

If the index is not available or if the chunk ID is not in the index (or the index is out-of-date), *dsync* tries to locate and hash files that are most likely to contain chunks that the receiver is trying to fetch. Note that the disk is used as an optimization: *dsync* validates all chunks found by indexing to ensure that the transfer remains correct even if the index is out-of-date or corrupt.

dsync requires CPU resources to compute the hash of data from the local disk and to perform scheduling computations that determine which chunks it should fetch from which resource.

3.2 Control Loop

Figure 1 depicts the main *dsync* control loop. *dsync* performs two basic tasks (shown with thicker lines in the figure) at every scheduling step in its loop: First, *dsync* determines which resources are available. Second, given the set of available resources, *dsync* schedules *operations* on those resources. An operation (potentially) returns a chunk. After assigning some number of operations to available resources, *dsync* executes the operations. This control loop repeats until *dsync* fetches all of the chunks, and is invoked every time an operation completes. Sections 4 and 5 provide more detail about the above tasks.

4 Avoiding Contention

dsync’s first task is to decide which resources are available for use: the network, the disk, or both. Deciding which resources to use (and which not to use) is critical because using a busy resource might contend with the ongoing transfer and might actually slow down the transfer.² For example, searching the local disk for useful chunks might delay *dsync* from writing chunks to disk if those chunks are received over a fast network. Spending CPU time to compute a more optimal schedule, or doing so too often, could delay reading data from the network or disk.

Back-pressure is the central mechanism that *dsync* uses to detect contention and decide which resources are available to use. *dsync* receives back-pressure information from the network and the disk through the explicit communication of queue sizes.³ While the disk exposes information to *dsync* indicating if it is busy, the network indicates how many requests it can handle without incurring unnecessary queuing. *dsync* also monitors its CPU consumption to determine how much time it can spend computing a schedule.

²Note that *dsync* tries to avoid resource contention within an ongoing transfer. Global resource management between *dsync* and other running processes currently remains the responsibility of the operating system, but is an interesting area for future work.

³Our experience agrees with that of others who have found that explicit queues between components makes it easier to behave gracefully under high load [23, 2].

Disk: *dsync* learns that the disk is busy from two queues:

- **Writer process queue:** *dsync* checks for disk back-pressure by examining the outstanding data queued to the writer process (a separate process that writes completed files to disk). If the writer process has pending writes, *dsync* concludes that the disk is overloaded and stops scheduling new disk read operations. (Writes are critical to the transfer process, so they are prioritized over read operations.)
- **Incoming network queue:** *dsync* also infers disk pressure by examining the incoming data queued in-kernel on the network socket. If there is a queue build-up, *dsync* stops scheduling disk read operations to avoid slowing the receiver.

Network: *dsync* tracks availability of network resources by monitoring the *outstanding data requests queue* on a per sender basis. If any sender's outstanding data queue size is less than the number of bytes the underlying network connection can handle when all the scheduled network operations are issued, *dsync* infers that the network is available.

CPU: *dsync* consumes CPU to hash the content of local files it is searching, and to perform the (potentially expensive) scheduling computations described in the following section. When searching the local filesystem, the scheduler must decide which of thousands of candidate files to search for which of thousands of chunks. Thus, *dsync* ensures that performing expensive re-computation of a new, optimized schedule using CPU resources does not slow down an on-going transfer using the following guidelines at every scheduling step.

- As it does for disk contention, *dsync* defers scheduling computations if there is build-up in the *incoming network queue*.
- To reduce the amount of computation, *dsync* only re-computes the schedule if more than 10% of the total number of chunks or number of available operations have changed. This optimization reduces CPU consumption while bounding the maximum estimation error (e.g., limiting the number of chunks that would be erroneously requested from the network when a perfectly up-to-date schedule would think that those chunks would be better served from the local disk).
- *dsync* limits the total time spent in recomputation to 10% of the current transfer time.
- Finally, the scheduler limits the total number of files that can be outstanding at any time. This limitation sacrifices some opportunities for optimization in favor of reducing the time spent in recomputation.

The experiments in Section 7 show the importance of avoiding resource contention when adapting to different environments. For example, on a fast network (gigabit LAN), failing to respond to back pressure means that disk and CPU contention impose up to a 4x slowdown vs. receiving only from the network.

5 Scheduling Resources

Once the control loop has decided which resources are available, *dsync*'s scheduler decides which operations to perform on those resources. The actual operations are performed by a separate plugin for each resource, as described in the next subsections. The scheduler itself takes the list of outstanding chunks and the results of any previously completed network and disk operations and determines which operations to issue in two steps. (A) First, the scheduler decides which disk operations to perform. (B) Second, given an ordering of disk operations, the scheduler decides which chunks to request from the network. At a high level, *dsync*'s scheduler opportunistically uses the disk to speed the transfer, knowing that it can ultimately fetch all chunks using the network.

This two-step operation is motivated by the different ways these resources behave. Storage provides an opportunity to accelerate network operations, but the utility of each disk operation is uncertain *a priori*. In contrast, the network can provide all needed chunks. A natural design, then, is to send to the disk plugin those operations with the highest estimated benefit, and to request from the network plugin those chunks that are least likely to be found on disk.

5.1 Resource Operations

Network operations are relatively straightforward; *dsync* instructs the network plugin to fetch a set of chunks (given the hashes) from the network. Depending on the configuration, the network plugin gets the chunks from the sender directly and/or from peers, if they exist.

Disk operations are more complex because *dsync* may not know the contents of files in the local file system. *dsync* can issue three operations to the disk plugin to find useful chunks on the disk.

CACHE operations access the chunk index (a collection of $\langle \text{ChunkID}, \text{ChunkSize}, \text{filename}, \text{offset}, \text{file metadata} \rangle$ -tuples), if one exists. **CACHE-CHECK** operations query to see if a chunk ID is in the index. If it is, *dsync* then `stat()`s the identified file to see if its metadata matches that stored in the index. If it does not, *dsync* invalidates all cache entries pointing to the specified file. Otherwise, *dsync* adds a potential **CACHE-READ** operation, which reads the chunk from the specified file and offset, verifies that it matches the given hash, and returns the chunk data.

HASH operations ask the disk plugin to read the specified file, divide it into chunks, hash each chunk to determine a chunk ID, compare each chunk ID to a list of needed chunk IDs, and return the data associated with any matches via a callback mechanism. Found chunks are used directly, and *dsync* avoids fetching them from the network. The identifiers and metadata of all chunks are cached in the index.

STAT operations cause the disk plugin to read a given directory and `stat()` each entry therein. STAT does not locate chunks by itself. Instead, it generates additional potential disk operations for the scheduler, allowing *dsync* to traverse the filesystem. Each file found generates a possible HASH operation, and each directory generates another possible STAT operation.

dsync also increases the estimated priority of HASHing the enclosing file from a CACHE-CHECK operation as the number of CACHE-READs from that file increase. This optimization reduces the number of disk seeks when reading a file with many useful chunks (Section 7.4).

5.2 Step A: Ordering Disk Operations

The scheduler estimates, for each disk operation, a cost to benefit ratio called the *cost per useful chunk delivered* or *CPC*. The scheduler estimates the costs by dynamically monitoring the cost of operations such as disk seeks or hashing chunks. The scheduler estimates the benefit from these operations using an extensible set of heuristics. These heuristics consider factors such as the similarity of the source filename and the name of the file being hashed, their sizes, and their paths in the filesystem, with files on the receiver’s filesystem being awarded a higher benefit if they are more similar to the source file. The scheduler then schedules operations greedily by ascending CPC.⁴

The *CPC* for an operation, *OP*, is defined as:

$$\begin{aligned} CPC_{OP} &= \frac{\text{total cost of the operation}}{\text{expected num. of useful chunks delivered}} \\ &= T_{OP}/E[N_{delivered}] \end{aligned}$$

T_{OP} , the *total* operation cost, includes all costs associated with delivering the chunk to *dsync* memory, including the “wasted” work associated with processing non-matching chunks. T_{OP} is thus the sum of the up-front operation cost (`OpCost()`) and the transfer cost (`XferCost()`). Operation cost is how long the disk plugin expects an operation to take. Transfer cost is how long the disk plugin expects the transfer of all chunks to *dsync* to take. $E[N_{delivered}]$ is the expected number of useful chunks found and is calculated by estimating the probability (p) of finding each required chunk when a particular disk operation is issued. $E[N_{delivered}]$ is then calculated as the sum of the probabilities across all source chunks.

⁴Determining an optimal schedule is NP-hard (Set Cover), and we’ve found that a greedy schedule works well in practice.

We present the details of computing the CPC for each disk operation in Section 6.2, along with the heuristics used to estimate the benefit of operations.

Metadata Groups

Schedule computation can involve thousands of disk operations and many thousands of chunks, with the corresponding scheduling decisions involving millions of *chunks * disk operations* pairs. Because the scheduler estimates the probability of finding a chunk using only metadata matching, all chunks with the same metadata have the same probabilities. Often, this means that all chunks from a particular source file are equivalent, unless those chunks appear in multiple source files. *dsync* aggregates these chunks in *metadata groups*, and performs scheduling at this higher granularity.

Metadata-group aggregation has two benefits. First, it greatly reduces the complexity and memory required for scheduling. Second, metadata groups facilitate making large sequential reads and writes by maintaining their associated chunks in request order. The receiver can then request the chunks in a group in an order that facilitates high-performance reads and writes.

5.3 Step B: Choosing Network Operations

After sorting the chunks by *CPC*, the scheduler selects which chunks to request from the network by balancing three issues:

- Request chunks that are unlikely to be found soon on disk (either because the operations that would find them are scheduled last or because the probability of finding the chunks is low).
- Give the network plugin enough chunks so that it can optimize their retrieval (by allocating chunks to sources using the rarest-random strategy [10]).
- Request chunks in order to allow both the sender and receiver to read and write in large sequential blocks.

The scheduler achieves this by computing the expected time of arrival (ETA) for each chunk C_j . If the chunk is found in the first operation, its ETA is simply the cost of the first operation. Otherwise, its cost must include the cost of the next operation, and so on, in descending *CPC* order through each operation:

$$\begin{aligned} T_i &= \text{Total cost of operation } i \\ ETA_j &= T_0 + (1 - p_{0,j}) \cdot T_1 + (1 - p_{0,j}) \cdot (1 - p_{1,j}) \cdot T_2 \\ &\quad + \dots + \prod_i (1 - p_{i,j}) \cdot T_m \end{aligned}$$

where $p_{i,j}$ is the probability of finding chunk j using operation at position i .

The scheduler then prioritizes chunk requests to the network by sending those with the highest ETA first.

Operation Type	Op-Path	Cost	T_{OP}	a.txt: p	b.txt: p	CPC
Time = 0						
STAT	/d/new	1	9	1.000	1.000	1.125
Time = 1						
HASH	/d/new/a.txt	4	4	1.000	0.350	0.741
STAT	/d	1	9	0.444	0.444	2.533
HASH	/d/new/core	64	64	0.007	0.007	1142.9
Expected Time to Arrival →				4	418.4	

Table 1: Disk operation optimization example.

5.4 Optimization Example

Table 1 shows *dsync*’s optimization for a sample transfer. The sender wants to transfer two 64 KB files, `a.txt` and `b.txt`, to a destination path on the receiver named `/d/new`. The destination path has two files in it: `a.txt`, which is an exact copy of the file being sent; and a 1 MB `core` file. We assume the operation cost of `STAT` and `CHUNK-HASH` (cost for reading and hashing one chunk) to be 1 unit. At the first scheduling decision (Time = 0), the only available operation is a `STAT` of the target directory. The probability of locating useful chunks via this `STAT` operation is set to unity since we expect a good chance of finding useful content in the target directory of the transfer.

After performing this operation (Time = 1), the scheduler has the three listed options. The cost of the first option, `HASH`ing `/d/new/a.txt`, is 4 (OpCost(`CHUNK_HASH`) times the number of chunks in the file—64 KB / 16 KB chunk size). Because `a.txt` matches everything about the source file, *dsync* sets the probability of finding chunks from it to 1. We then compute the probability of finding chunks for file `b.txt` from the operation `HASH`(`/d/new/a.txt`) by comparing the metadata of files `a.txt` and source file `b.txt`; Both files belong to the same target directory, have the same size, but are only a file type match (both are text files), and do not have the same modification time. We give each of these factors a suitable weight to arrive at $p = 1 \cdot 1 \cdot 0.5 \cdot 0.7 = 0.350$ (Details in Section 6.2). The *CPC* for the operation is then computed as $T_{OP} = 4$ divided by $E[N_{delivered}](4 * 1.0 + 4 * 0.35)$. The costs for subsequent operations are computed similarly. The disk operations are then sorted as per their *CPC*. Intuitively, since hashing `a.txt` found on the disk appears more beneficial for the `.txt` files in the on-going transfer, its *CPC* is lower than the other operations.

Given the *CPC* ordering of operations, the expected time of arrival is then computed. Once again, because the probability of finding the chunks in `a.txt` during the first `HASH` operation (at Time = 0) is 1, the ETA for chunks in `a.txt` is simply equal to the cost of that first

operation. Thus, because the chunks in `b.txt` have a higher ETA from the disk, the scheduler will schedule those chunks to the network.

6 Implementation

This section provides details of (1) the software components and (2) the *CPC* parameters used in the current implementation of *dsync*.

6.1 Software Components

dsync is implemented using the Data-Oriented Transfer (DOT) service [21]. The *dsync_client* application transfers a file or file tree by first sending one or more files to its local DOT service daemon, receiving an object ID (OID) for each one. *dsync_client* creates a tree descriptor from these OIDs and the associated file metadata and sends the tree descriptor to the receiving *dsync_client*. The receiving *dsync_client* requests the OIDs from its local DOT daemon, which fetches the corresponding files using various DOT transfer plugins. Finally, *dsync_client* re-creates the file tree in the destination path.

The receiver side of *dsync* is implemented as a client binary and a DOT “transfer plugin”. DOT transfer plugins receive chunk requests from the main DOT system. The bulk of *dsync*’s functionality is a new DOT transfer plugin (called *dsync*), which contains the scheduler component that schedules chunk requests as described earlier, and a second plugin (*xdisk*) that searches for hashes in local files. For point-to-point transfers, *dsync* uses the DOT default transfer plugin that provides an RPC-based chunk request protocol. To properly size the number of outstanding requests, we modified this transfer plugin to send with each chunk the number of bytes required to saturate the network connection. This implementation requires kernel socket buffer auto-tuning, obtaining the queue size via `getsockopt`.

For peer-to-peer transfers, *dsync* uses DOT’s existing Similarity-Enhanced Transfer (SET) network plugin [19]. SET discovers additional network peers using a centralized tracker. These network peers could be either seeding the entire file or could be downloading the file at the same time as the receiver (swarmers). To fetch chunks from swarmers more efficiently, SET exchanges bitmaps with the swarmers to learn of their available chunks. Finally, once it knows which chunks are available from which sources, SET requests chunks from these sources using the rarest-random strategy.

For *dsync* to work well on high-speed networks, we added a new zero-copy sending client interface mechanism to DOT to reduce data copies: `put(filename)`; clients using this mechanism must ensure that the file is not modified until the transfer has completed. DOT still stores additional copies of files at the receiver.

In addition to this interface, we improved DOT’s efficiency in several ways, particularly with its handling of large (multi-gigabyte) files; as a result, our performance results differ somewhat from those described earlier [21].

6.2 Disk Operation CPC Parameters

For the implementation of the scheduling framework described in Section 5, we treat CACHE, HASH, and STAT operations individually.

6.2.1 CPC for CACHE Operations

There are two types of CACHE operations: CACHE-READ and CACHE-CHECK.

$CPC_{CACHE-READ}$: $OpCost(CACHE-READ)$ involves the overhead in reading and hashing a single chunk. *dsync* tracks this per chunk cost ($OpCost(CHUNK_HASH)$) across all disk operations; when operations complete, the scheduler updates the operation cost as an exponentially weighted moving average (EWMA) with parameter $\alpha = 0.2$, to provide a small amount of smoothing against transient delays but respond quickly to overall load changes.

CACHE-READ operations do not incur any transfer cost since the chunk is already in memory. Since one CACHE-READ operation results in a single chunk, the expected number of useful chunks delivered is optimistically set to 1.

$CPC_{CACHE-CHECK}$: $OpCost(CACHE-CHECK)$ involves the overhead in looking up the chunk index. Since the chunk index is an in-memory hash table, we assume this cost to be negligible. Further, a CACHE-CHECK operation does not result in chunks; it results in more CACHE-READ operations. Thus, the ratio $\frac{XferCost()}{E[N_{delivered}]}$ is set to the CPC of the best possible CACHE-READ operation. As seen above, this value equals $OpCost(CHUNK_HASH)$.

6.2.2 CPC for HASH Operation

$OpCost(HASH)$ is $OpCost(CHUNK_HASH)$ multiplied by the number of chunks in the file. $XferCost(HASH)$ is zero since the chunks are already in memory (they were just hashed). To calculate the expected number of useful chunks that can result from hashing a file, the scheduler calculates for each requested chunk, the probability that the chunk will be delivered by hashing this file (p).

The scheduler uses an extensible set of heuristics to estimate p based on the metadata and path similarity between candidate file to be HASHed and the source file that a requested chunk belongs to specified for the transfer. While these simple heuristics have been effective on our test cases, our emphasis is on creating a framework that can be extended with additional parameters rather than on determining absolute values for the parameters below. Note that we need only generate a good relative ordering

of candidate operations; determining the true probabilities for p is unnecessary. Currently, our heuristics use the parameters described below to compute p , using the formula:

$$p = p_{mc} \cdot p_{fs} \cdot p_{ss} \cdot p_{tm} \cdot p_{pc}$$

Max chunks: p_{mc} . If the candidate file is smaller than the source file, it can contain at most a $\frac{size_{candidate}}{size_{source}}$ fraction of the chunks from the source file; thus, any given chunk can be found with probability at most $p_{mc} = \frac{\min(size_{candidate}, size_{source})}{size_{source}}$.

Filename similarity: p_{fs} . An identical filename provides $p_{fs} = 1$ (i.e., this file is very likely to contain the chunk). A prefix or suffix match gives $p_{fs} = 0.7$, a file type match (using filename extensions) provides $p_{fs} = 0.5$, and no match sets $p_{fs} = 0.1$.

Size similarity: p_{ss} . As with filenames, an exact size match sets $p_{ss} = 1$. If the file size difference is within 10%, we set $0.5 \leq p_{ss} \leq 0.9$ as a linear function of size difference; otherwise, $p_{ss} = 0.1$.

Modification time match: p_{tm} . Modification time is a weak indicator of similarity, but a mismatch does not strongly indicate *dissimilarity*. We therefore set $p_{tm} = 1$ for an exact match, $0.9 \leq p_{tm} \leq 1$ for a fuzzy match within 10 seconds, and $p_{tm} = 0.7$ otherwise.

Path commonality: p_{pc} . Our final heuristic measures the directory structure distance between the source and candidate files. Intuitively, the source path `/u/bob/src/dsync` is more related to the candidate path `/u/bob/src/junk` than to `/u/carol/src/junk`, but the candidate path `/u/carol/src/dsync` is *more* likely to contain relevant files than either junk directory. The heuristic therefore categorizes matches as leftmost, rightmost, or middle matches.

The match coefficient of two paths is the ratio of the number of directory names in the longest match between the source and candidate paths over the maximum path length. How should p_{pc} vary with the length of the match? A leftmost or middle match reflects finding paths farther away in the directory hierarchy. The number of potential candidate files grows roughly exponentially with distance in the directory hierarchy, so a leftmost or middle match is assigned a p_{pc} that decreases exponentially with the match coefficient. A rightmost match, however, reflects that the directory structures have re-converged. We therefore assign their p_{pc} as the actual match coefficient.

Finally, if a chunk belongs to multiple source files, we compute its probability as the maximum probability computed using each file it belongs to (a conservative lower bound):

$$p = \max(p^{file1}, \dots, p^{fileN})$$

6.2.3 CPC for STAT Operation

The scheduler tracks $\text{OpCost}(\text{STAT})$ similar to $\text{OpCost}(\text{CHUNK_HASH})$. For STAT operations, the chunk probabilities are calculated using path commonality only, $p = p_{pc}$ and thus $E[N_{delivered}]$ can be computed. Once again, STATs do not find chunks; they produce more HASH operations. Thus, the ratio $\frac{\text{XferCost}()}{E[N_{delivered}]}$ for a STAT is set to the CPC of the *best* possible HASH operation that could be produced by that STAT. (In other words, the scheduler avoids STATs that could not possibly generate better HASH operations than those currently available).

Before the STAT is performed, the scheduler knows only the directory name and nothing about the files contained therein. The best possible HASH operation from a given STAT therefore would match perfectly on max chunks, file name, size, and modification time ($p_{mc} = p_{fs} = p_{ss} = p_{tm} = 1$). This lower bound cost is optionally weighted to account for the fact that most STATs do not produce a best-case HASH operation.

$$\begin{aligned} \text{best } p &= p_{pc} \cdot 1 \cdot 1 \cdot 1 \cdot 1 \\ \text{CPC}_{\text{STAT}} &= w \cdot \left(\frac{\text{OpCost}(\text{STAT})}{\sum p_{pc}} + \frac{\text{OpCost}(\text{CHUNK_HASH})}{\max(\text{best } p)} \right) \end{aligned}$$

7 Evaluation

In this section, we evaluate whether *dsync* achieves the goal set out in Section 2: Does *dsync* correctly and efficiently transfer files under a wide range of conditions? Our experiments show that it does:

- *dsync* effectively uses local and remote resources when initial states of the receivers are varied.
- *dsync* ensures that transfers are never slowed. *dsync*'s back-pressure mechanisms avoid overloading critical resources (a) when the network is fast, (b) when the disk is loaded, and (c) by not bottle-necking senders due to excessive outstanding requests.
- *dsync*'s heuristics quickly locate useful similar files in a real filesystem; *dsync* intelligently uses a pre-computed index when available.
- *dsync* successfully provides the best of both *rsync* and multi-source transfers, quickly synchronizing multiple machines that start with different versions of a target directory across 120–370 PlanetLab nodes using a real workload.

7.1 Method

Environment. We conduct our evaluation using Emulab [24] and PlanetLab [18]. Unless otherwise specified,

Emulab nodes are “pc3000” nodes with 3 GHz 64-bit Intel Xeon processors, 2 GB RAM, and two 146 GB 10,000 RPM SCSI hard disk drives running Fedora Core 6 Linux. We choose as many reachable nodes as possible for each PlanetLab experiment, with multiple machines per site. The majority of PlanetLab experiments used 371 receivers. The sender for the data and the peer tracker for the PlanetLab experiments are each quad-core 2 GHz Intel Xeon PCs with 8 GB RAM located at CMU.

Software. We compare *dsync*'s performance to *rsync* and CoBlitz [16] as a baseline. (We compare with CoBlitz because its performance already exceeds that of BitTorrent.) Achieving a fair comparison is complicated by the different ways these systems operate. *dsync* first computes the hash of the entire object before communicating with the receiver, while *rsync* pipelines this operation. We run *rsync* in server mode, where it sends data over a clear-text TCP connection to the receiver. By default, *dsync* uses SSH to establish its control channel, but then transfers the data over a separate TCP connection.

To provide a fair comparison, in all experiments, we measure the time *after* putting the object into *dsync* on the sender and establishing the SSH control channel, and we ensure that *rsync* is able to read the source file from buffer cache. This comparison is not perfect—it equalizes for disk read times, but *dsync* would normally pay a cost for hashing the object in advance. In addition *dsync* maintains a cache of received chunks, which puts extra pressure on the buffer cache. Hence, in most of our evaluation, we focus on the relative scaling between the two; in actual use the numbers would vary by a small amount.

To compare with CoBlitz, our clients contact a CoBlitz deployment across as many nodes as are available. (CoBlitz ran in a new, dedicated slice for this experiment.) We use CoBlitz in the out-of-order delivery mode. To illustrate *dsync*'s effectiveness, we also compare *dsync* to basic DOT, SET [19], *dsync*-1src and *dsync*-NoPressure when applicable. Basic DOT neither exploits local resources nor multiple receivers. SET is a BitTorrent-like swarming technique that does not use local resources. *dsync*-1src is simply *dsync* that does not use swarming. *dsync*-NoPressure is *dsync* configured to ignore back-pressure.

Finally, we start PlanetLab experiments using 50 parallel SSH connections. At most, this design takes two minutes to contact each node. All the experiment results are averaged over three runs. In the case of CoBlitz, the average is computed over the last three runs in a set of four runs to avoid cache misses. We log transfer throughput for each node, defined to be the ratio of the total bytes transferred to the transfer time taken by that node.

Scenarios. In many of our experiments, we configure the local filesystem in one of the following ways:

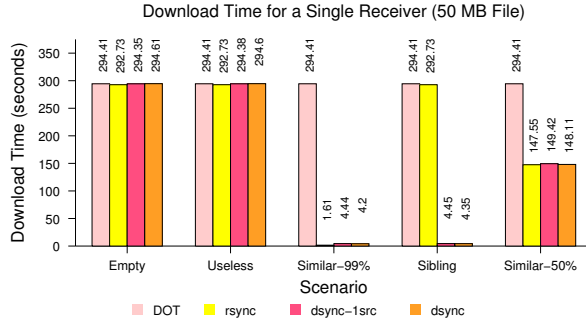


Figure 2: Transfer time for a single receiver using five initial filesystem configurations over a 1.544 Mbps link.

- Empty: The destination filesystem is empty, affording no opportunities to find similar files.
- Useless: The destination filesystem has many files, but none similar to the desired file. The filesystem is large enough that *dsync* does not completely explore it during a transfer.
- Similar: The destination filesystem contains a file by the same name as the one being transferred in the final destination directory. We set the similarity of this file with the file being transferred to either 50% or 99%.
- Sibling: The destination filesystem contains a 99% similar file with the same name but in a directory that is a sibling to the final destination directory.

7.2 *dsync* works in many conditions

In this section, we explore *dsync*'s performance under a variety of transfer scenarios, varying the number of receivers, the network conditions, and the initial filesystem configurations.

7.2.1 Single receiver

We first use a simple 50 MB data transfer from one source to one receiver to examine whether *dsync* correctly finds and uses similar files to the one being transferred and to evaluate its overhead relative to standard programs such as *rsync*. The two nodes are connected via a 1.544 Mbps (T1) network with 40 ms round-trip delay. Figure 2 compares the transfer times using the original DOT system, *dsync-1src*, *dsync*, and *rsync* under the five filesystem scenarios outlined above. This figure shows three important results:

***dsync* rapidly locates similar files.** In the “Similar-99%” configuration, both *dsync* and *rsync* locate the nearly-identical file by the same name and use it to rapidly complete the data transfer. In the “Sibling” configuration, only *dsync*⁵ locates this file using its search heuristics.

⁵*rsync* supports a manual Sibling mode where the user must point *rsync* at the relevant directory.

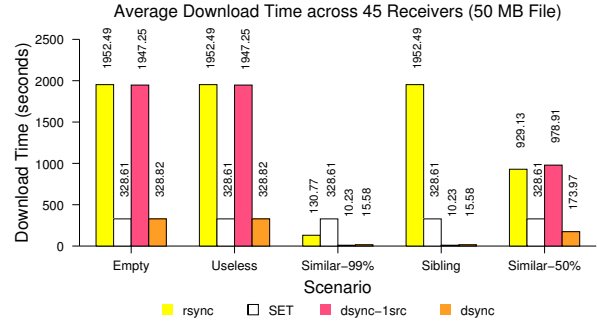


Figure 3: Transfer time across 45 nodes using five filesystem configurations over 1.544 Mbps links.

***dsync*'s performance is as good as its subsystems individually.** When transferring over the network, *dsync* is as fast as the DOT system on which it builds. *dsync* incurs negligible penalty over *dsync-1src* when using both disk and swarming plugins.

***dsync* adds only small overhead vs. *rsync*.** In the similar-99 case, *dsync* requires 2.8 seconds more transfer time than *rsync*. The overhead over the basic time to read and chunk (2.3 seconds) comes from a mix of the DOT system requesting chunk hashes *before* reading from either the network or the local filesystem (0.4 seconds), *dsync*'s post-hash index creation (0.3 seconds) and an artifact of the Linux kernel sizing its TCP buffers too large, resulting in an excess queue for *dsync* to drain before it can fetch the last chunk from the single source (250 KB at 1.5 Mbps = 1.4 seconds).

7.2.2 Multiple receivers, homogeneous initial states

We next explore *dsync*'s ability to swarm among multiple receivers. In this Emulab experiment, we use a single 10 Mbps sender and 45 receivers with T1 links downloading a 50MB file. All the receivers have homogeneous initial filesystem configurations. The receivers use a mix of hardware, ranging from pc650 to pc3000. The sender is a pc850. Figure 3 shows the average and standard deviation of the transfer times for *rsync*, SET, *dsync-1src* and *dsync* as the initial configurations are varied.

***dsync* effectively uses peers.** The “Empty” and the “Useless” scenarios show the benefit from swarming – In *rsync* and *dsync-1src*, all the receivers obtain the data from a single sender, dividing its 10 Mbps among themselves. *dsync* (and SET) swarm among the receivers to almost saturate their download capacities, resulting in 6× speedup.

***dsync* effectively combines peering and local resources.** In the “Similar-99%” scenario, *dsync* is 21× faster than SET because it uses the local file. *dsync* is also 8× faster than *rsync* in this scenario, because the single *rsync* sender becomes a computational bottleneck.

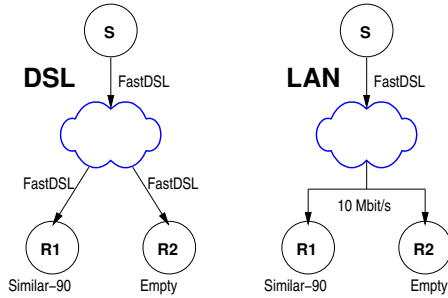


Figure 4: Network and initial states of R1 and R2.

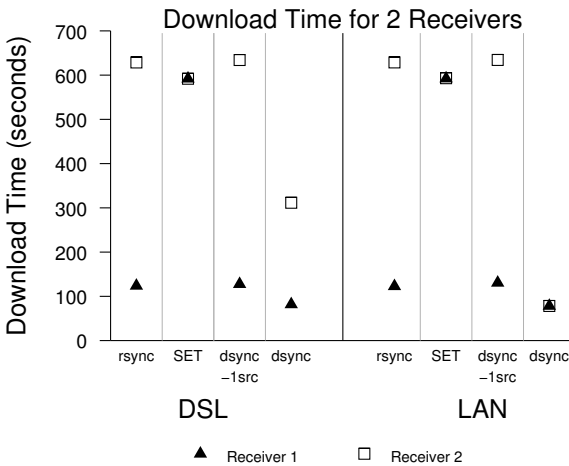


Figure 5: Transfer time in a two-receiver transfer.

Finally, in the “Similar-50%” scenario, all schemes except SET effectively use the local data to double their throughput.

7.2.3 Multiple receivers, heterogeneous initial states

This experiment shows how *dsync* can synchronize multiple machines whose disks start in different initial configurations from each other by intelligently using a combination of similar local files, the original sender, and the other recipients. Figure 4 depicts two scenarios, and Figure 5 shows the download time for a 50 MB file with *rsync*, SET, *dsync-1src*, and *dsync*.

In both *rsync* and *dsync-1src*, Receiver 1 uses the similar file on its local disk to speed the transfer while Receiver 2 fetches the entire file from the source. Because the sender is the bottleneck in both cases, the DSL and LAN cases perform similarly. SET alone behaves like other peer-to-peer swarming systems; Receiver 1 does not use the local file. Instead, the two receivers collectively fetch one copy of all of the file data from the sender. Receiver 2’s download is faster with SET than with *rsync* because with *rsync*, the clients fetch 1.10 copies (100% and 10%, respectively) of the data; SET’s rarest-first heuristic ensures that the receivers do not request the same chunks.

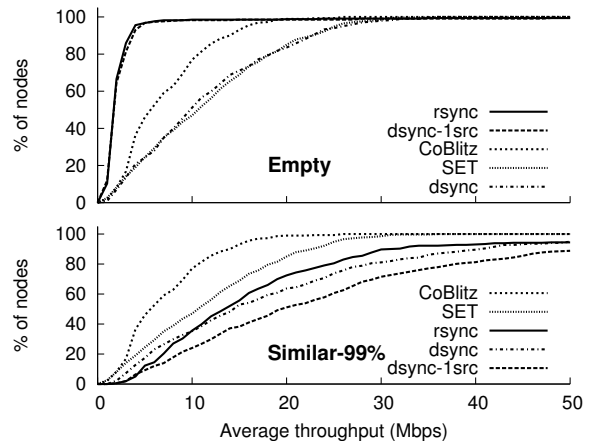


Figure 6: CDF of average throughput across 371 PlanetLab nodes for different filesystem configurations.

dsync shows drastic improvement from searching the local disk *and* downloading from a peer. Receiver 1 fetches most of the file from the local disk and offers it to Receiver 2. In the DSL scenario, Receiver 2 fetches the file from both the sender and Receiver 1, transferring the file about twice as fast as *rsync* can. In the LAN scenario, Receiver 2 grabs almost all of the bytes over its fast link to Receiver 1, and the two receivers fetch only the missing 10% of the data from the slow source. Overall, *dsync* automatically uses the best resources to download the file, using the disk when possible and transferring only one copy of the remaining data over the network.

7.2.4 Multiple receivers in the wild (PlanetLab)

We now explore the performance of *rsync*, CoBlitz, SET, *dsync-1src* and *dsync* when all available PlanetLab nodes simultaneously download a 50 MB file. All the nodes are in identical initial filesystem configurations.

Figure 6 shows the throughput for all the schemes in the “Empty” scenario. The single sender becomes a bottleneck in *rsync* and *dsync-1src*. CoBlitz uses multi-source downloads and caching to outperform *rsync*. Somewhat unexpectedly, *dsync* outperforms CoBlitz; the median *dsync* node is $1.67\times$ faster. The most evident reason for this is that *dsync* receivers make more aggressive use of the original source, drawing 1.5 GB from the sender vs. CoBlitz’s 135 MB. Thus, CoBlitz is more sender friendly. However, CoBlitz’s performance is impaired as the remaining senders are extremely busy PlanetLab nodes. *dsync*’s load balancing makes more effective (but perhaps less friendly) use of the fast source. *dsync*’s performance is again comparable to that of the underlying SET transfer mechanism in the “Empty” scenario: suggesting that *dsync*’s filesystem searching does not slow down the transfers even on busy nodes.

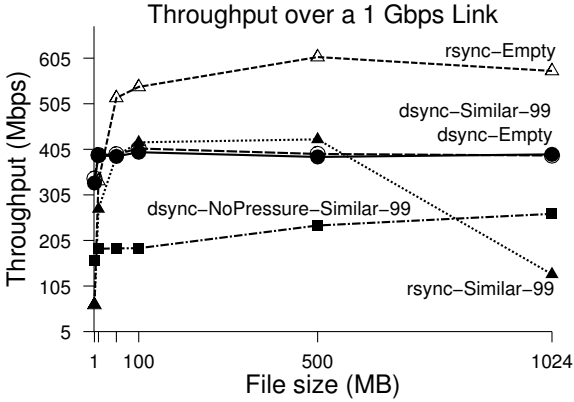


Figure 7: Throughput over a 1 Gbps link vs. file size.

In the “Similar-99%” scenario, *dsync* automatically uses local resources when appropriate to improve the median throughput by 50% over “Empty” scenario. An interesting note about this figure is that *dsync-1src* (no swarming) is slightly *faster*, because the additional network load appeared to slow down the PlanetLab nodes. This effect did not appear in similar tests on Emulab; we accept this as a case in which *dsync*’s generality made it perform slightly worse in one particularly extreme example. *dsync* did, however, correctly reduce network traffic in these cases.

7.3 *dsync*’s back-pressure is effective

This section shows that *dsync* effectively uses its back-pressure mechanisms to adapt to overloaded critical resources in order to retain high performance.

7.3.1 Gigabit network

In this experiment, we transfer files varying from 1 MB to 1 GB over a 1 Gbps network between two nodes. This network is faster than the disk, which can sustain only 70 MB/sec, or 550 Mbps. Thus, the disk is the overloaded critical resource that limits transfer speed. Figure 7 shows throughput as file size increases for the various systems.

***dsync* uses back-pressure to defer disk operations.** *dsync* correctly notices that the network is faster than the disk and ignores the local file. Thus, *dsync-Similar-99%* and *dsync-Empty* perform similarly. *rsync-Similar-99%*, however, suffers significantly when it tries to use a 1 GB local file because it competes with itself trying to write out the data read from the disk.

It is noteworthy that *dsync* outperforms *rsync*-similar even though the prototype *dsync* implementation’s overall throughput is lower than *rsync-Empty* as a result of additional data copies and hashing operations in the underlying DOT system. This figure also shows that *dsync*’s improvements come mostly from its back-pressure mechanisms and not simply from using both the network and

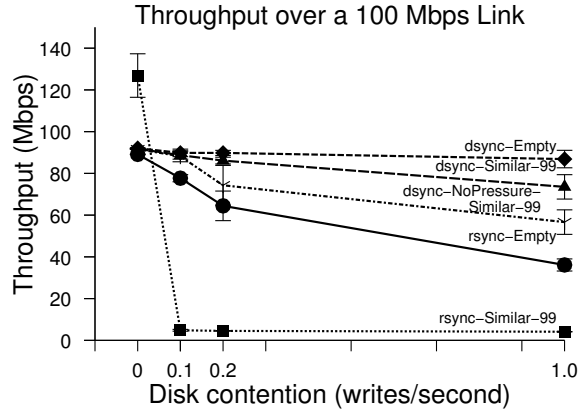


Figure 8: Throughput vs. disk contention for a 1 GB file.

the disk. While *dsync-NoPressure* also slightly outperforms *rsync* in Similar-99% case, its performance is much lower than that of the pressure-aware *dsync*.

7.3.2 Slow disk

The gigabit case is not a rare exception to be handled by an “if” statement. Critical resources can be overloaded in a variety of scenarios. For example, we simulate slow disk scenarios using a process performing bulk writes (1 MB at a time) to disk and varying the frequency of writes. The single sender and the receiver are connected by 100 Mbps LAN and transfer a 1 GB file. Figure 8 shows the throughput as disk load increases.

***dsync* again uses back-pressure to defer disk operations.** *rsync-Similar-99%*, however, still tries to read from the overloaded local disk, and its performance degrades much more rapidly than fetching from the network alone.

7.3.3 Dynamic adaptation to changing load

Figure 9 shows a transfer in the Slow Disk scenario (above), in which we introduce disk load 20 seconds into the transfer. Each plot shows the number of bits/second read from each resource during the transfer.

***dsync* dynamically adapts to changing load conditions among resources.** We observe that *dsync* backs off from reading from the disk when the contention starts. This helps *dsync* to finish writing the transferred file more quickly (101 seconds) than *dsync-NoPressure* (117 seconds).

7.4 *dsync* effectively uses the local disk

This section highlights *dsync*’s ability to discover useful data on disk regardless of index state. *dsync* uses a pre-computed index when available, and its heuristics for data discovery otherwise.

***dsync* correctly decides between index reads and sequential reads.** To evaluate *dsync*’s use of the index, we transfer files between 10 MB and 1 GB between two

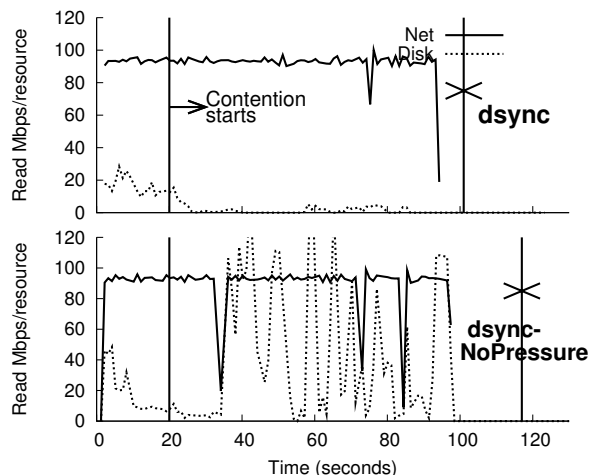


Figure 9: Throughput as disk contention starts 20 seconds into transfer. X indicates transfer completion.

	10	50	100	1024
<i>dsync</i>	0.94	4.39	7.75	167
<i>dsync-naive</i>	0.91	4.10	8.63	525

Table 2: Transfer time (seconds) from intelligent use of proactive index as file size is varied from 10 MB to 1 GB.

nodes. The receiver maintains a pre-computed index of the files on the disk, and all files being transferred are available on the receiver’s disk. *dsync-naive* satisfies the cache hit for each chunk request by opening the relevant file, seeking to the chunk offset and reading the required bytes. *dsync*, however, observes that several chunks are desired from the same file and sequentially reads the file to serve all cache hits. The results of this test are shown in Table 2. As the file size increases, this strategy is more effective—a 1 GB file is copied 3× faster.

***dsync*’s filesystem search heuristics are effective** at exploring the disk to identify useful chunks. In this experiment, the sender wants to transfer a file called “a.out” into the destination path “/s/y/n/c/test/” on the receiver. Each set of rows in Table 3 represents iterations of the experiment with the useful file in a different directory. We measure the number of STAT and HASH operations and the number of megabytes hashed up to and including when *dsync* hashed the useful file. We disabled fetching chunks from the network since our intent was to measure the discovery process of the heuristics and not transfer time. We compare our heuristics with two simple strategies: (a) An allstats strategy where we “stat” the entire hard disk and hash the files in the descending order of their usefulness, and (b) a breadth-first strategy (BFS) that traverses the directory hierarchy by hashing all files in its sub-tree before going to its parent.

The root of the filesystem is the desktop machine home directory of one of the authors. The file system had 445,355 files in 37,157 directories, occupying 76 GB. The alternate paths we explore represent a few feasible locations in which a duplicate copy of a file might exist in a home directory. We do not claim that this set is representative of other filesystems, but it demonstrates that the heuristics perform well in these (reasonable) scenarios. We performed no tuning of the heuristics specific to this filesystem, and never used it as a test during development.

In the top group, the useful file is in the destination directory. Here, the heuristics find the file quickly and do not hash any other files beforehand. When the useful file has a completely different name “diff”, *dsync* issues STAT requests of several other directories before hashing it. In the second group, the useful file is in a sibling directory to the destination path. Here, *dsync*’s heuristics perform several additional STATs before finding the useful file, but relatively few additional hashes. In fact, the scheduler suggests only one extraneous HASH operation for a file with size match. BFS performs reasonably well in scenarios 1-6. Allstats incurs a constant 56.17 seconds to stat the entire filesystem before hashing the useful file.

The final group in the table shows the results when the useful file is in a completely unrelated directory on the receiver’s file system. Here, when the file name matches exactly, *dsync* STATs almost 15% of the directories in the file system, but only hashes two extraneous files. The last line of the table reveals an interesting result: when the useful file has neither a name or path match, *dsync* inventoried the entire file system looking for useful files, but still only hashed 175 MB of unnecessary data. This results in performance similar to allstats. BFS, however, degrades as it hashes ~4 GB of data. Thus, *dsync* intelligently adapts to different filesystem configurations unlike simple strategies.

7.5 Real workload

dsync substantially improves throughput using a workload obtained from our day-to-day use—running our experiments required us to frequently update PlanetLab with the latest *dsync* code. This workload consisted of 3 binaries (totaling 27 MB): the DOT daemon (gtcd), a client (gcp) and the disk read process (aiod). We choose one such snapshot during our experimentation where we synchronized 120 PlanetLab nodes. A third of these nodes did not have any copy of our code, another third had a debug copy in a sibling directory and the last third had almost the latest version in the destination directory. We also repeated this experiment for 371 nodes.

Figure 10 shows the CDF of average throughput across these nodes for *rsync*, SET and *dsync*. Using *dsync*, the median node synchronizes 1.4-1.5× faster than it does using SET, and 5× faster than using *rsync*.

Path of Useful File	Size, Similarity (MB, %)	<i>dsync</i>			<i>BFS</i>		
		STAT Ops	Files (MB) Hashed	Seconds	STAT Ops	Files (MB) Hashed	Seconds
1. <i>/s/y/n/c/test/a.out</i>	12.0, 100%	1 (0%)	1 (12)	0.30	1 (0%)	8 (70.05)	1.25
2. <i>/s/y/n/c/test/a.out</i>	12.1, 11%	1 (0%)	1 (12.1)	0.29	1 (0%)	8 (70.05)	1.25
3. <i>/s/y/n/c/test/diff</i>	12.0, 100%	373 (1%)	1 (12)	0.76	1 (0%)	8 (70.05)	1.25
4. <i>/s/y-old/n/c/test/a.out</i>	12.0, 100%	879 (2.37%)	2 (24.03)	1.84	364 (0.97%)	3212 (139.4)	4.17
5. <i>/s/y-old/n/c/test/a.out</i>	12.1, 11%	879 (2.37%)	2 (24.03)	1.84	364 (0.97%)	3212 (139.4)	4.17
6. <i>/s/y-old/n/c/test/diff</i>	12.0, 100%	2225 (5.98%)	2 (24.03)	4.67	364 (0.97%)	3212 (139.4)	4.17
7. <i>/a.out</i>	12.0, 100%	6038 (16.25%)	3 (34.03)	8.90	6962 (18.74%)	51,128 (4002.4)	1771.9
8. <i>/a.out</i>	12.1, 11%	6038 (16.25%)	3 (34.03)	8.90	6962 (18.74%)	51,128 (4002.4)	1771.9
9. <i>/diff</i>	12.0, 100%	37,157(100%)	227 (173.8)	68.3	6962 (18.74%)	51,128 (4002.4)	1771.9

Table 3: Performance of *dsync* heuristics on a real directory tree. Files with 11% similarity are an old version of “a.out”.

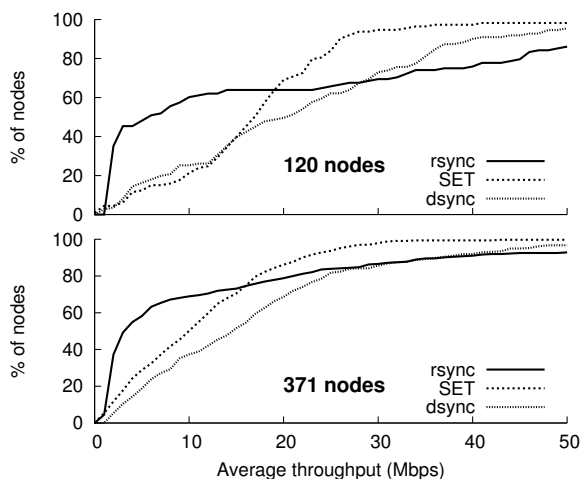


Figure 10: CDF of average throughput when *dsync* is used for software distribution on Planetlab.

8 Related Work

Our work fits into the broad scope of systems that leverage an excess of one resource to compensate for the lack of another. Examples range from the familiar use of caching to avoid computation to more contemporary systems that use spare CPU cycles, in the form of speculative execution, to hide disk latency [3] and network latency [15]; or that mask network latency or limited capacity by explicit caching or prefetching [9]. *dsync* specifically uses local computation and storage resources to replace network bandwidth, when such a trade-off is appropriate.

Like many such systems, ensuring that *dsync* does not create a scarcity of a local resource is key to performing well under a wide range of conditions. These problems do not just occur with disk contention; for example, the benefits of Web prefetching can be eclipsed if the prefetching traffic slows down foreground traffic, a situation that can again be solved by strictly prioritizing traffic that is guaranteed to be useful [9].

Many systems address the problems of drawing from multiple sources that have heterogeneous performance. The Blue File System [14], for example, selects sources

from a group of storage devices based on power and performance characteristics. The River cluster filesystem [2] shares our goal of automatically using sources of different rates (though in its case, different disk rates within a cluster). Like *dsync*, River does so by managing the request queue to each source. Exposing the queues between components helps systems such as SEDA shed load intelligently under overload conditions [23], much as *dsync*’s queue monitoring avoids local resource contention.

As we noted in the introduction, many of the techniques that *dsync* uses to obtain chunks are borrowed from systems such as *rsync* [22], LBFS [13], CFS [6], and Shark [1]. A particularly noteworthy relative is Shotgun, which used the Bullet’ content distribution mesh to disseminate the diff produced by *rsync* batch mode [11]. This approach works well—it avoids the overhead of multiple filesystem traversals on the sender and greatly reduces the amount of network bandwidth used—but *rsync* batch mode requires that the recipients be in identical starting states and is subject to the same limitations as *rsync* in finding only a single file to draw from.

dsync uses a peer-to-peer content distribution system to allow nodes to swarm with each other. For implementation convenience, we used the SET plugin from DOT, but we did not use any of its unique features—our system could use any transfer system that allows transferring chunks independent of a file. In keeping with our philosophy of opportunistic resource use and broad applicability, the most likely alternative distribution system for *dsync* is CoBlitz [16] since it is cache-based and does not require that sources “push” data into a distribution system.

Finally, several forms of system support for chunk-based indexing provide possible opportunities for improving the timeliness and efficiency of the pre-computed index. Linux’s `inotify()` would at least permit an index daemon to re-hash files when they change. A content-addressable filesystem would avoid the need for an external index at all, provided its chunking and hashing was *dsync*-friendly. The CZIP [17] approach of adding user-level chunk index headers to files would permit *dsync* to

much more rapidly examine new candidate files during its filesystem exploration.

9 Conclusion

dsync is a data transfer system that correctly and efficiently transfers files under a wide range of operating conditions. *dsync* effectively uses all available resources to improve data transfer performance using a novel optimization framework. With an extremely fast sender and network, *dsync* transfers all data over the network instead of transferring the data more slowly from the receiver's local disk; with a slow network, it will aggressively search the receiver's disk for even a few chunks of useful data; and when sending to multiple receivers, *dsync* will use peer receivers as additional data sources.

While combining network, disk-search, and peer-to-peer techniques is conceptually simple, doing so while ensuring that resource contention does not impair performance is difficult. The keys to *dsync*'s success include adaptively deciding *whether* to search the local disk, intelligently scheduling disk search operations and network chunk requests to minimize the total transfer time, and constantly monitoring local resource contention. Making good scheduling decisions also requires that *dsync* deal with practical system issues such as ensuring large sequential read/write operations, that can be hidden behind the content-based naming abstraction.

Our evaluation shows that *dsync* performs well in a wide range of operating environments, achieving performance near that of existing tools on the workloads for which they were designed, while drastically outperforming them in scenarios beyond their design parameters.

Acknowledgments

We thank the anonymous reviewers who provided valuable feedback on this work; Shimin Chen for discussion about database optimization strategies; Phil Gibbons and Anupam Gupta for discussions about the set cover problem; and KyoungSoo Park and Vivek Pai for their help running CoBlitz. This work was supported in part by NSF CAREER award CNS-0546551 and DARPA Grant HR0011-07-1-0025.

References

- [1] S. Annapureddy, M. J. Freedman, and D. Mazières. Shark: Scaling file servers via cooperative caching. In *Proc. 2nd USENIX NSDI*, May 2005.
- [2] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick. Cluster I/O with river: Making the fast case common. In *Proc. IOPADS: Input/Output for Parallel and Distributed Systems*, May 1999.
- [3] F. Chang and G. Gibson. Automatic I/O hint generation through speculative execution. In *Proc. 3rd USENIX OSDI*, Feb. 1999.
- [4] B. Cohen. Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems*, June 2003.
- [5] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making backup cheap and easy. In *Proc. 5th USENIX OSDI*, Dec. 2002.
- [6] 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)*, Oct. 2001.
- [7] T. E. Denehy and W. W. Hsu. Duplicate Management for Reference Data. Research Report RJ10305, IBM, Oct. 2003.
- [8] F. Douglis and A. Iyengar. Application-specific delta-encoding via resemblance detection. In *Proceedings of the USENIX Annual Technical Conference*, June 2003.
- [9] R. Kokku, P. Yalagandula, A. Venkataramani, and M. Dahlin. NPS: A non-interfering deployable web prefetching system. In *Proc. 4th USENIX USITS*, Mar. 2003.
- [10] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proc. 19th ACM SOSP*, Oct. 2003.
- [11] 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*, Apr. 2005.
- [12] J. C. Mogul, Y. M. Chan, and T. Kelly. Design, implementation, and evaluation of duplicate transfer detection in HTTP. In *Proc. 1st USENIX NSDI*, Mar. 2004.
- [13] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2001.
- [14] E. B. Nightingale and J. Flinn. Energy-efficiency and storage flexibility in the blue file system. In *Proc. 6th USENIX OSDI*, pages 363–378, Dec. 2004.
- [15] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. In *Proc. 20th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2003.
- [16] K. Park and V. Pai. Scale and Performance in the CoBlitz Large-File Distribution Service. In *Proc. 3rd USENIX NSDI*, May 2006.
- [17] K. Park, S. Ihm, M. Bowman, and V. S. Pai. Supporting practical content-addressable caching with CZIP compression. In *Proceedings of the USENIX Annual Technical Conference*, June 2007.
- [18] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the Internet. In *Proc. 1st ACM Hotnets*, Oct. 2002.
- [19] H. Pucha, D. G. Andersen, and M. Kaminsky. Exploiting similarity for multi-source downloads using file handprints. In *Proc. 4th USENIX NSDI*, Apr. 2007.
- [20] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.
- [21] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil. An architecture for Internet data transfer. In *Proc. 3rd USENIX NSDI*, May 2006.
- [22] A. Tridgell and P. Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, Department of Computer Science, The Australian National University, Canberra, Australia, 1996.
- [23] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2001.
- [24] 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, Dec. 2002.