A Low-bandwidth Network File System

Athicha Muthitacharoen, Benjie Chen, and David Mazières
MIT Laboratory for Computer Science and NYU Department of Computer Science
{athicha,benjie}@lcs.mit.edu, dm@cs.nyu.edu

Abstract

Users rarely consider running network file systems over slow or wide-area networks, as the performance would be unacceptable and the bandwidth consumption too high. Nonetheless, efficient remote file access would often be desirable over such networks—particularly when high latency makes remote login sessions unresponsive. Rather than run interactive programs such as editors remotely, users could run the programs locally and manipulate remote files through the file system. To do so, however, would require a network file system that consumes less bandwidth than most current file systems.

This paper presents LBFS, a network file system designed for low-bandwidth networks. LBFS exploits similarities between files or versions of the same file to save bandwidth. It avoids sending data over the network when the same data can already be found in the server's file system or the client's cache. Using this technique in conjunction with conventional compression and caching, LBFS consumes over an order of magnitude less bandwidth than traditional network file systems on common workloads.

1 Introduction

This paper describes LBFS, a network file system designed for low-bandwidth networks. People typically run network file systems over LANs or campus-area networks with 10 Mbit/sec or more bandwidth. Over slower, wide-area networks, data transfers saturate bottleneck links and cause unacceptable delays. Interactive programs freeze, not responding to user input during file I/O, batch commands can take many times their normal execution time, and other less aggressive network applications are starved for bandwidth. Users must therefore employ different techniques to accomplish what over the LAN they would do through the file system.

People often have occasion to work over networks slower than LANs. Even with broadband Internet access, a person working from home usually only has a fraction of a Mbit/sec of upstream bandwidth. A company with offices in several cities may have many users collaborating over a single 1.5 Mbit/sec T1 line. A consultant constantly traveling between various sites may want to sit down and access the same project files from every location.

In the absence of a network file system, people generally resort to one of two methods of accessing remote data. They either make and edit local copies of files, running the risk of an update conflict, or else they use remote login to view and edit files in place on another machine. If the network has long latency, remote login is particularly frustrating as interactive applications are slow in responding to user input. Worse yet, many graphical applications, such as figure editors and postscript previewers, consume too much bandwidth to run practically over the wide-area network.

Network file systems have the potential to alleviate the inconveniences associated with remote data access. In addition to offering the interface people already prefer for local-area networks, a file system can provide tight consistency, avoiding the problem of conflicts when two people update the same file. File systems can also better tolerate network latency than remote login sessions. By running interactive programs locally and accessing remote data through a file system, one avoids the overhead of a network round trip on every user input event.

To be practical over the wide-area network, however, a file system must consume significantly less bandwidth than most current file systems, both to maintain acceptable performance and to avoid monopolizing network links in use for other purposes. Unfortunately, application writers commonly assume that file I/O will be no slower than a megabyte or so per second. For instance, an interactive editor will stop to write out 100 KByte "auto-save" files without worrying about delaying a user's typing or consuming significant resources. A traditional file system transmits the entire con-

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Space and Naval Warfare Systems Center, San Diego, under contract N66001-00-1-8927.

tents of such files over the network, blocking the editor for the duration of the transfer. In contrast, LBFS often transmits far less data than applications write, greatly reducing the time spent waiting for file I/O.

To reduce its bandwidth requirements, LBFS exploits cross-file similarities. Files written out by applications often contain a number of segments in common with other files or previous versions of the same file. Auto-save files are only one example. Object files output by compilers, temporary files used by the RCS revision control system, postscript files, and word processing documents often contain substantial similarity from one revision to the next. Any copying or concatenation of files, such as when building program libraries out of object files, also leads to significant duplication of contents.

To exploit these inter-file similarities, the LBFS file server divides the files it stores into chunks and indexes the chunks by hash value. The LBFS client similarly indexes a large persistent file cache. When transferring a file between the client and server, LBFS identifies chunks of data that the recipient already has in other files and avoids transmitting the redundant data over the network. In conjunction with conventional compression, this technique saves over an order of magnitude of communications bandwidth on many common workloads.

LBFS provides traditional file system semantics and consistency. Files reside safely on the server once closed, and clients see the server's latest version when they open a file. Thus, LBFS can reasonably be used in place of any other network file system without breaking software or disturbing users. Other file systems have dealt with slow and even intermittent network connectivity by relaxing file system consistency. These techniques largely complement LBFS's, and could be combined with LBFS for even greater bandwidth savings. However, since altered semantics may not be suitable for all purposes, we chose to focus on reducing bandwidth to see just how much we could save without changing accepted consistency guarantees.

The next section describes related work. Section 3 gives LBFS's algorithm for finding commonality between files and explains how the LBFS protocol takes advantage of it. Section 4 describes the implementation of LBFS. Section 5 shows how effective LBFS's technique for compressing file traffic can be. Finally, Section 6 concludes.

2 Related Work

Past projects have attacked the problem of network file systems on slow networks from several angles. LBFS complements most previous work. Because it provides consistency and does not place significant hardware or file system structure requirements on the server, LBFS's approach can unobtrusively be combined with other techniques to get additional savings in network bandwidth.

A number of file systems have properties that help them tolerate high network latency. AFS [9] uses server callbacks to inform clients when other clients have modified cached files. Thus, users can often access cached AFS files without requiring any network traffic. Leases [7] are a modification to callbacks in which the server's obligation to inform a client of changes expires after a certain period of time. Leases reduce the state stored by a server, free the server from contacting clients who haven't touched a file in a while, and avoid problems when a client to which the server has promised a callback has crashed or gone off the network. The NFS4 protocol [20] reduces network round trips by batching file system operations. All of the above techniques are applicable to LBFS. In fact, LBFS currently uses leases and a large, persistent cache to provide AFS-like close-to-open consistency.

Many file systems use write-behind to tolerate latency. Echo [14] performs write-behind of metadata operations, allowing immediate completion of operations that traditionally require a network round trip. In JetFile [8], the last machine to write a file becomes the file's server, and can transmit its contents directly to the next reader.

The CODA file system [10] supports slow networks and even disconnected operation. Changes to the file system are logged on the client and written back to the server in the background when there is network connectivity. To implement this functionality, CODA provides weaker-than-traditional consistency guarantees. It allows update conflicts, which users may need to resolve manually. CODA saves bandwidth because it avoids transferring files to the server when they are deleted or overwritten quickly on the client. LBFS, in contrast, simply reduces the bandwidth required for each file transfer. Thus, LBFS could benefit from CODA-style deferred operations, and CODA could benefit from LBFS file transfer compression.

Bayou [18] further investigates conflict resolution for optimistic updates in disconnected systems, but unlike CODA, it does not provide a file system. Rather, Bayou supplies an API with which to implement application-specific merging and conflict resolution. OceanStore [2] applies Bayou's conflict resolution mechanisms to a file system and extends it to work with untrusted servers that only ever see data in encrypted format. TACT [25] explores the spectrum between absolute consistency and Bayou's weaker model.

Lee et. al. [12] have extended CODA to support operation-based updates. A proxy-client strongly connected to the server duplicates the client's computation in the hopes of duplicating its output files. Users run a modified shell that bundles up commands for the proxy-client to reexecute. Using forward error correction, the client and proxy-client can even patch up small glitches in the output files, such as different dates. When successful, operation-based updates deliver a tremendous bandwidth savings. However, the technique is fairly complementary to LBFS. LBFS works well with interactive applications such as editors that would be hard to reexecute on a proxy-client. Operation-based updates can reduce communications bandwidth with command-line utilities such as image converters for which LBFS offers no savings. Operation-based updates require a dedicated proxy-client machine, making them a bit cumbersome to set up. Perhaps for this reason the technique is not in widespread use by any file system today.

Spring and Wetherall have proposed a protocol-independent technique for eliminating redundant network traffic [21]. They assume two cooperating caches at either end of a slow network link. Both caches store identical copies of the last n Megabytes of network traffic (for values of n up to 100). When one end must send data that already exists in the cache, it instead sends a token specifying where to find the data in the cache. To identify redundant traffic, the two ends index cache data by 64-byte anchors [13], randomly chosen based on hash value. When data to be sent has a 64-byte anchor in common with previous traffic, the matching region is expanded in both directions to elide the greatest amount of data. LBFS's approach is similar in spirit to the Spring and Wetherall technique. However, LBFS supports multiple clients accessing the file file system and even local users changing the file system underneath the server. Thus, it cannot assume that the client and server have identical state.

Rsync [23] copies a directory tree over the network onto another directory tree containing similar files—typically from a previous version of the same tree. Rsync saves bandwidth by exploiting commonality between files. The problem is similar to synchronizing a client's file cache with the server or vice versa. In fact, Tridgell suggests applying rsync to a file system in his thesis. Though rsync was one of the inspirations for LBFS, file caching in real time is somewhat different from directory tree mirroring. LBFS thus uses a different algorithm. We discuss the rsync algorithm in more detail and compare it to our approach in Section 3.1.

A number of Unix utilities operate on differences between files. `diff` computes the difference between two text files. `patch` applies the output of `diff` to transform one file into the other. There have been studies of the problem of describing one file in terms of a minimal set of edits to another [22]. Mogul et. al. [17] have investigated transmitting such deltas to save bandwidth when updating cached web pages. The CVS [1] version management system ships patches over the network to bring a user's working copy of a directory tree up to date. Unlike CVS, however, a file system cannot store a complete revision history for all files. Therefore, the LBFS server will typically not have an exact old version from which to compute differences.

3 Design

LBFS is designed to save bandwidth while providing traditional file system semantics. In particular, LBFS provides close-to-open consistency. After a client has written and closed a file, another client opening the same file will always see the new contents. Moreover, once a file is successfully written and closed, the data resides safely at the server. These semantics are similar to those of AFS. Other work exploring relaxed consistency semantics may apply to LBFS, but we wished to build a file system that could directly substitute for a widely accepted network file system in use today.

To save bandwidth, LBFS uses a large, persistent file cache at the client. LBFS assumes clients will have enough cache to contain a user's entire working set of files (a reasonable assumption given the capacities of cheap IDE disks today). With such aggressive caching, most client–server communication is solely for the purpose of maintaining consistency. When a user modifies a file, the client must transmit the changes to the server (since in our model the client might crash or be cut from the network). Similarly, when a client reads a file last modified by a different client, the server must send it the latest version of the file.

LBFS reduces bandwidth requirements further by exploiting similarities between files. When possible, it reconstitutes files using chunks of existing data in the file system and client cache instead of transmitting those chunks over the network. Of course, not all applications can benefit from this technique. A worst case scenario is when applications encrypt files on disk, since two different encryptions of the same file have no commonality whatsoever. Nonetheless, LBFS provides significant bandwidth reduction for common workloads.

For the remainder of this section, we first discuss the issues involved in indexing chunks of the file system and cache data. We describe the advantages and disadvantages of several approaches, including LBFS's particular solution. Then, we describe the actual LBFS protocol and its use of chunk indexes.

3.1 Indexing

On both the client and server, LBFS must index a set of files to recognize data chunks it can avoid sending over the network. To save chunk transfers, LBFS relies on the collision-resistant properties of the SHA-1 [6] hash function. The probability of two inputs to SHA-1 producing the same output is far lower than the probability of hardware bit errors. Thus, LBFS follows the widely-accepted practice of assuming no hash collisions. If the client and server both have data chunks producing the same SHA-1 hash, they assume the two are really the same chunk and avoid transferring its contents over the network.

The central challenge in indexing file chunks to identify commonality is keeping the index a reasonable size while dealing with shifting offsets. As an example, one could index the hashes of all aligned 8 KByte data blocks in files. To transfer a file, the sender would transmit only hashes of the file's blocks, and the receiver would request only blocks not

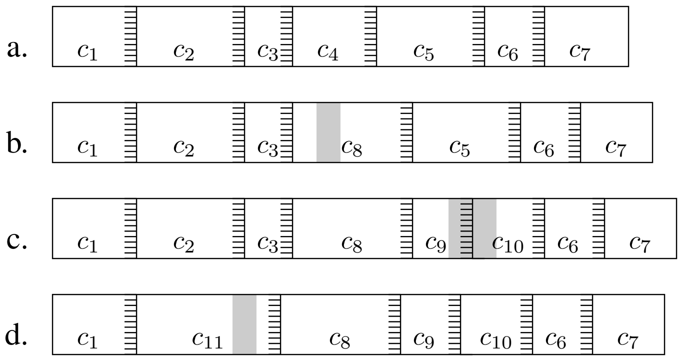

Figure 1: Chunks of a file before and after various edits. Horizontal stripes show 48-byte regions with magic hash values creating chunk boundaries. Gray shading shows regions of the file that were changed by an edit.

already in its database. Unfortunately, a single byte inserted at the start of a large file would shift all the block boundaries, change the hashes of all the file's blocks, and thereby thwart any potential bandwidth savings.

As an alternative, one might index files by the hashes of all (overlapping) 8 KByte blocks at all offsets. Such a scheme would require storage many times the size of the indexed files (almost one index entry per byte of file data). The size in itself might be tolerable given large enough disks, but every file modification might require thousands of index insertions. The cost of performing so many updates to an index in secondary storage would be prohibitive.

Rsync more practically tackles this problem by considering only two files at a time. When transferring file F from machine A to machine B, if B already has a file F' by the same name, rsync guesses the two files may be similar and attempts to exploit that fact. A simplified version of the rsync algorithm proceeds as follows. First, the recipient, B, breaks its file F' into non-overlapping, contiguous, fixed-size blocks. B transmits hashes of those blocks to A. A in turn begins computing the hashes of all (overlapping) blocks of F. If any of those hashes matches one from F', A avoids sending the corresponding sections of F, instead telling B where to find the data in F'.

A couple of complications arise when trying to apply the rsync algorithm to a file system, however. First, rsync's choice of F' based on filename is too simple. For example, when editing file `foo`, emacs creates an auto-save file named `#foo#`. RCS uses even less suggestive temporary file names such as `_1v22825`. Thus, the recipient would have to choose F' using something other than file names. It might select F' based on a fixed-size "sketch" of F, using Broder's resemblance estimation technique [4]. However, even ignoring the additional cost of this approach, sometimes F can best be reconstructed from chunks of multiple files—consider `ar`, which outputs software libraries containing many object files.

3.1.1 LBFS Solution

In order to use chunks from multiple files on the recipient, LBFS takes a different approach from that of rsync. It considers only non-overlapping chunks of files and avoids sensitivity to shifting file offsets by setting chunk boundaries based on file contents, rather than on position within a file. Insertions and deletions therefore only affect the surrounding chunks. Similar techniques have been used successfully in the past to segment files for the purpose of detecting unauthorized copying [3].

To divide a file into chunks, LBFS examines every (overlapping) 48-byte region of the file and with probability 2^{-13} over each region's contents considers it to be the end of a data chunk. LBFS selects these boundary regions—called *breakpoints*—using Rabin fingerprints [19]. A Rabin fingerprint is the polynomial representation of the data modulo a predetermined irreducible polynomial. We chose fingerprints because they are efficient to compute on a sliding window in a file. When the low-order 13 bits of a region's fingerprint equal a chosen value, the region constitutes a breakpoint. Assuming random data, the expected chunk size is $2^{13} = 8192 = 8$ KBytes (plus the size of the 48-byte breakpoint window). As will be discussed in Section 5.1, we experimented with various window sizes and found that 48 bytes provided good results (though the effect of window size was not huge).

Figure 1 shows how LBFS might divide up a file and what happens to chunk boundaries after a series of edits. **a.** shows the original file, divided into variable length chunks with breakpoints determined by a hash of each 48-byte region. **b.** shows the effects of inserting some text into the file. The text is inserted in chunk c_4, producing a new, larger chunk c_8. However, all other chunks remain the same. Thus, one need only send c_8 to transfer the new file to a recipient that already has the old version. Modifying a file can also change the number of chunks. **c.** shows the effects of inserting data that contains a breakpoint. Bytes are inserted in c_5, splitting that chunk into two new chunks c_9 and c_{10}. Again, the file can be transfered by sending only the two new chunks. Finally, **d.** shows a modification in which one of the breakpoints is eliminated. Chunks c_2 and c_3 of the old file are now combined into a new chunk, c_{11}, which must be transmitted to compose the new file.

3.1.2 Pathological Cases

Unfortunately, variable-sized chunks can lead to some pathological behavior. If every 48 bytes of a file happened to be a breakpoint, for instance, the index would be as large as the file. Worse yet, hashes of chunks sent over the wire would consume as much bandwidth as just sending the file. Conversely, a file might contain enormous chunks. In particular, the Rabin fingerprint has the property that a long extent of zeros will never contain a breakpoint. As discussed later in Section 3.2, LBFS transmits the contents of a chunk in the body of an RPC message. Having arbitrary size RPC messages would be somewhat inconvenient, since most RPC libraries hold messages in memory to unmarshal them.

To avoid the pathological cases, LBFS imposes a minimum and maximum chunk size. The minimum chunk size is 2K. Any 48-byte region hashing to a magic value in the first 2K after a breakpoint does not constitute a new breakpoint. The maximum chunk size is 64K. If the file contents does not produce a breakpoint every 64K, LBFS will artificially insert chunk boundaries. Such artificial suppression and creation of breakpoints can disrupt the synchronization of file chunks between versions of a file. The risk, if this occurs, is that LBFS will perform no better than an ordinary file system. Fortunately, synchronization problems most often result from stylized files—for instance a long run of zeros, or a few repeated sequences none of which has a breakpoint—and such files do well under conventional compression. Since all LBFS RPC traffic gets conventionally compressed, pathological cases do not necessarily translate into slow file access.

3.1.3 Chunk Database

LBFS uses a database to identify and locate duplicate data chunks. It indexes each chunk by the first 64 bits of its SHA-1 hash. The database maps these 64-bit keys to $\langle \text{file}, \text{offset}, \text{count} \rangle$ triples. This mapping must be updated whenever a file is modified. Keeping such a database in sync with the file system could potentially incur significant overhead. Moreover, if files exported through LBFS are modified by other means—for instance by a local process on the server—LBFS cannot prevent them from growing inconsistent with the database. Even on the client side, an inopportune crash could potentially corrupt the contents of the disk cache.

To avoid synchronization problems, LBFS never relies on the correctness of the chunk database. It recomputes the SHA-1 hash of any data chunk before using it to reconstruct a file. LBFS also uses the recomputed SHA-1 value to detect hash collisions in the database, since the 64-bit keys have a low but non-negligible probability of collision. Not relying on database integrity also frees LBFS from the need to worry about crash recovery. That in turn saves LBFS from making expensive synchronous database updates. The worst a corrupt database can do is degrade performance.

3.2 Protocol

The LBFS protocol is based on NFS version 3 [5]. NFS names all files by server-chosen opaque handles. Operations on handles include reading and writing data at specific offsets. LBFS adds extensions to exploit inter-file commonality during reads and writes. Most NFS clients poll the server on file open to check permissions and validate previously cached data. For recently accessed files, LBFS saves this round trip by adding leases to the protocol. Unlike many NFS clients, LBFS also practices aggressive pipelining of RPC calls to tolerate network latency. The system uses an asynchronous RPC library that efficiently supports large numbers of simultaneously outstanding RPCs. Finally, LBFS compresses all RPC traffic using conventional gzip compression.

3.2.1 File Consistency

The LBFS client currently performs whole file caching (though in the future we would like to cache only portions of very large files). When a user opens a file, if the file is not in the local cache or the cached version is not up to date, the client fetches a new version from the server. When a process that has written a file closes it, the client writes the data back to the server.

LBFS uses a three-tiered scheme to determine if a file is up to date. Whenever a client makes any RPC on a file in LBFS, it gets back a read lease on the file. The lease is a commitment on the part of the server to notify the client of any modifications made to that file during the term of the lease (by default one minute, though the duration is server-configurable). When a user opens a file, if the lease on the file has not expired and the version of the file in cache is up to date (meaning the server also has the same version), then the open succeeds immediately with no messages sent to the server.

If a user opens a file and the lease on the file has expired, then the client asks the server for the attributes of the file. This request implicitly grants the client a lease on the file. When the client gets the attributes, if the modification and inode change times are the same as when the file was stored in the cache, then the client uses the version in the cache with no further communication to the server. Finally, if the file times have changed, then the client must transfer the new contents from the server.

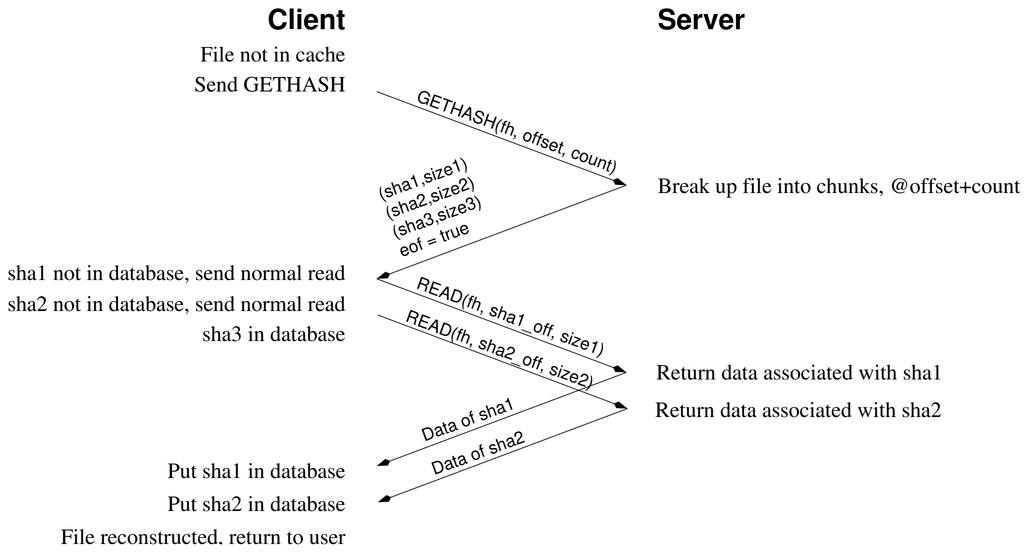

Figure 2: Reading a file using LBFS

Because LBFS only provides close-to-open consistency, a modified file does not need to be written back to the server until it is closed. Thus, LBFS does not need write leases on files—the server never demands back a dirty file. Moreover, when files are written back, they are committed atomically. Thus, if a client crashes or is cut from the network while writing a file, the file will not get corrupted or locked—other clients will simply continue to see the old version. When multiple processes on the same client have the same file open for writing, LBFS writes data back whenever any of the process closes the file. If multiple clients are writing the same file, then the last one to close the file will win and overwrite changes from the others. These semantics are similar to those of AFS.

3.2.2 File Reads

File reads in LBFS make use of one RPC procedure not in the NFS protocol, GETHASH.

GETHASH retrieves the hashes of data chunks in a file, so as to identify any chunks that already exist in the client's cache. GETHASH takes the same arguments as a READ RPC, namely a file handle, offset, and size (though in practice the size is always the maximum possible, because the client practices whole file operations). Instead of returning file data, however, GETHASH returns a vector of $\langle \text{SHA-1 hash}, \text{size} \rangle$ pairs.

Figure 2 shows the use of GETHASH. When downloading a file not in its cache, the client first calls GETHASH to obtain hashes of the file's chunks. Then, for any chunks not already in its cache, the client issues regular READ RPCs. Because the READ RPCs are pipelined, downloading a file generally only incurs two network-round trip times plus the cost of downloading any data not in the cache. For files larger than 1,024 chunks, the client must issue multiple GETHASH calls and may incur multiple round trips. However, network latency can be overlapped with transmission and disk I/O.

3.2.3 File Writes

File writes proceed somewhat differently in LBFS from NFS. While NFS updates files at the server incrementally with each write, LBFS updates them atomically at close time. There are several reasons for using atomic updates. Most importantly, the previous version of a file often has many chunks in common with the current version. Keeping the old version around helps LBFS exploit the commonality. Second, LBFS's file reconstruction protocol can significantly alter the order of writes to a file. Files being written back may have confusing intermediary states (for instance an ASCII file might temporarily contain blocks of 0s). Finally, atomic updates limit the potential damage of simultaneous writes from different clients. Since two clients writing the same file do not see each other's updates, simultaneously changing the same file is a bad idea. When this does occur, however, atomic updates at least ensure that the resulting file contains the coherent contents written by one of the clients, rather than a mishmash of both versions.

LBFS uses temporary files to implement atomic updates. The server first creates a unique temporary file, writes the temporary file, and only then atomically commits the contents to the real file being updated. While writing the temporary file, LBFS uses chunks of existing files to save bandwidth where possible. Four RPCs implement this update protocol: MKTMPFILE, TMPWRITE, CONDWRITE, and COMMITTMP.

MKTMPFILE creates a temporary file for later use in an atomic update. MKTMPFILE takes two arguments: first,

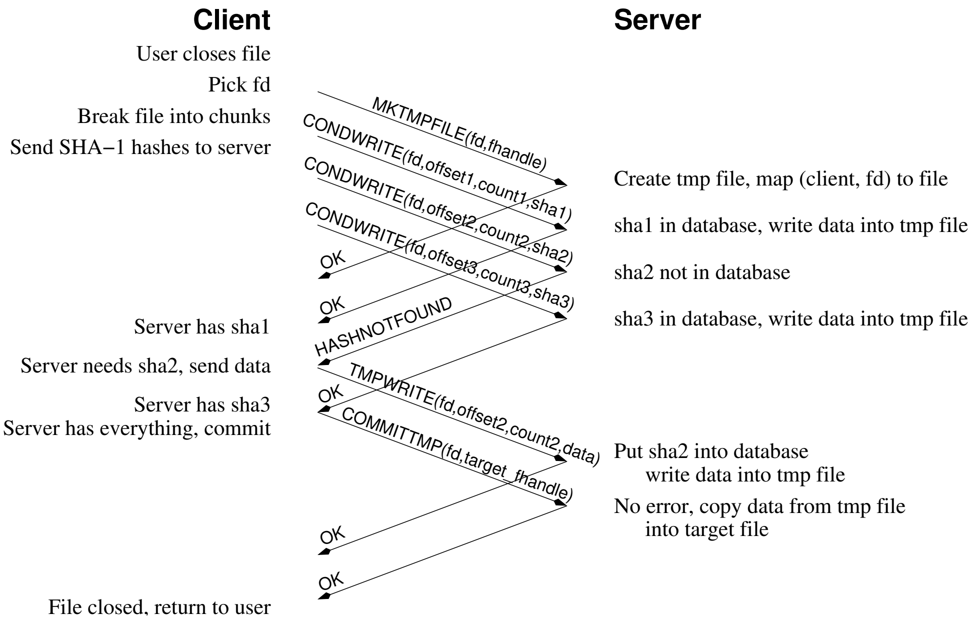

Figure 3: Writing a file using LBFS

the file handle of the file that will eventually be atomically updated, and second, a client-chosen "file descriptor" for the temporary file. After receiving the call, the server creates a temporary file in the same file system as the specified handle and keeps a mapping from the per-client file descriptor to the temporary file. Because clients choose the descriptors for temporary files, they can pipeline operations on temporary files before the MKTMPFILE RPC returns.

TMPWRITE is similar to a WRITE RPC. The only difference is the that a client-chosen temporary file descriptor replaces the NFS file handle in the arguments. An LBFS client sends TMPWRITEs instead of WRITEs to update a file created with MKTMPFILE.

CONDWRITE is similar to a TMPWRITE RPC. The arguments contain a file descriptor, offset, and length. Instead of the actual data to write, however, CONDWRITE arguments contain a SHA-1 hash of the data. If the server can find the data specified by the hash somewhere in its file system, it writes the data to the temporary file at the specified offset. If it cannot find the data, but the request would otherwise have completed, CONDWRITE returns the special error code HASHNOTFOUND.

COMMITTMP commits the contents of a temporary file to a permanent file if no error has occurred. It takes two arguments, a file descriptor for the temporary file, and a file handle for the permanent file. For each temporary file descriptor, the server keeps track of any errors other than HASHNOTFOUND that have occured during CONDWRITE and TMPWRITE RPCs. If any error has occurred on the file descriptor (e.g., disk full), COMMITTMP fails. Otherwise, the server replaces the contents of the target file with that of the temporary file and updates the chunk database to reflect the file's new contents. Since LBFS uses TCP, RPCs are delivered in order. Thus, the client can pipeline a COMMITTMP operation behind TMPWRITE RPCs.

Figure 3 shows the file write protocol in action. When a user closes a file that the client must write back, the client picks a file descriptor and issues a MKTMPFILE RPC with the handle of the closed file. In response, the server creates a temporary file handle and maps it to the specified file descriptor. The client then makes CONDWRITE RPCs for all data chunks in the file it is writing back. For any CONDWRITEs returning HASHNOTFOUND, the client also issues TMPWRITE calls. Finally, the client issues a COMMITTMP.

Pipelining of writes occurs in two stages. First, the client pipelines a series of CONDWRITE requests behind a MKTMPFILE RPC. Second, as the CONDWRITE replies come back, the client turns around and issues TMPWRITE RPCs for any HASHNOTFOUND responses. It pipelines the COMMITTMP immediately behind the last TMPWRITE. The communication overhead is therefore generally two round trip latencies, plus the transmission times of the RPCs. For large files, the client has a maximum limit on the number of outstanding CONDWRITE and TMPWRITE calls so as not to spend too much time sending calls when it can process replies. However, the extra network round trips will generally overlap with the transmission time of RPC calls.

3.2.4 Security Considerations

Because LBFS performs well over a wider range of networks than most file systems, the protocol must resist a

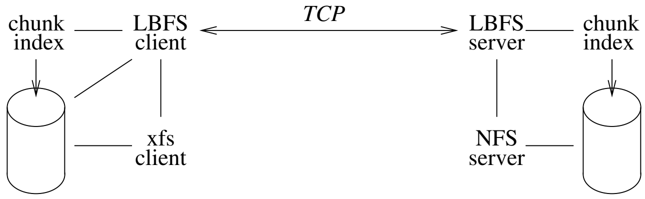

Figure 4: Overview of the LBFS implementation.

wider range of attacks. LBFS uses the security infrastructure from SFS [16]. Every server has a public key, which the client administrator specifies on the command line when mounting the server. In the future, we intend to embed public keys in pathnames as SFS does and to integrate LBFS into SFS's auto-mounting system so that unprivileged users on clients can access any server. The entire LBFS protocol, RPC headers and all, is passed through gzip compression, tagged with a message authentication code, and then encrypted. At mount time, the client and server negotiate a session key, the server authenticates itself to the user, and the user authenticates herself to the client, all using public key cryptography.

Finally, we note that LBFS may raise some non-network security issues. When several users share the same file system, LBFS could leak information about files a user is not allowed to read. Specifically, through careful use of CONDWRITE, a user can check whether the file system contains a particular chunk of data, even if the data resides in a read-protected file. Though CONDWRITE will fail on chunks the user cannot read, subtle timing differences may still let the user infer that the database contained the hash of the chunk. Nonetheless, LBFS should provide more than adequate security for most purposes, particularly given how widely users accept file systems that do not even encrypt network traffic.

4 Implementation

Figure 4 shows the architecture of the LBFS implementation. Both the client and server run at user-level. The client implements the file system using **xfs**, a device driver bundled with the ARLA [24] file system. The server accesses files through NFS. The client and server communicate over TCP, using Sun RPC. We used the asynchronous RPC library from the SFS toolkit [15] both for the server's NFS client and for LBFS client–server communication. The RPC library already had support for authenticating and encrypting traffic between a client and server. We added support for compression.

4.1 Chunk Index

The LBFS client and server both maintain chunk indexes, the server indexing file system contents and the client its local cache. The two share the same indexing code, implemented using the B-tree from SleepyCat software's BerkeleyDB package. Since LBFS never relies on chunk database correctness, it also does not concern itself with crash recoverability. LBFS avoids any synchronous database updates, and the server always replies to clients before inserting new chunks in its database. If the database loses a few hashes, clients will simply use more bandwidth until the database comes back up to date. There is a utility, `mkdb`, which builds a file system's database from scratch. However, if an LBFS server is run without a database, the server simply creates the database and populates it as users access files.

The one database operation on the critical path for clients is the lookup done as part of a CONDWRITE RPC. However, for all but the smallest files, CONDWRITEs are pipelined deeply enough to overlap database lookups with the transmission of any write data not found in the chunk index. For 8 KByte or smaller files, LBFS avoids CONDWRITEs and simply writes the files directly to the server in a single RPC. The overhead of multiple round trip times overshadows any potential bandwidth savings on such small files.

4.2 Server Implementation

Our main goal for the LBFS server implementation, other than saving bandwidth and providing acceptable performance, was to build a system that could unobtrusively be installed on an already running file system. This both isolates LBFS's benefits from physical file system layout and lets users take immediate advantage of LBFS on existing files without dedicating a disk or partition to it.

The LBFS server accesses the file system by pretending to be an NFS client, effectively translating LBFS requests into NFS. Building the LBFS server as an NFS client lets LBFS serve any file system for which an NFS server exists, which includes most file systems on most Unix operating systems.

Of course, the server might alternatively have been implemented using regular system calls to access the file system. However, NFS offers several advantages over the traditional system call interface. First, it simplifies the implementation, since the LBFS protocol is based on NFS. Second, NFS saved the LBFS server from the need to implement access control. The server simply maps LBFS requests to user IDs and tags the resulting NFS requests with those IDs, letting the NFS server decide whether or not to grant access. Finally, NFS allows the chunk index to be more resilient to outside file system changes. When a file is renamed, its NFS file handle remains the same and thus the chunk index does not need to be updated.

The LBFS server creates a "trash directory," `.lbfs.trash`, in the root directory of every file system it exports. The trash directory contains temporary files created by MKTMPFILE RPCs. As explained below, after a COMMITTMP RPC, the LBFS server does not delete the committed temporary file. Rather, if space is needed, it garbage-collects a random file in the trash directory. A background thread purges the database of pointers to deleted files.

4.2.1 Static i-number Problem

The one major disadvantage to using NFS is the lack of low-level control over file system data structures. In particular, Unix file system semantics dictate that a file's i-number not change when the file is overwritten. Thus, when the server commits a temporary file to a target file, it has to copy the contents of the temporary file onto the target file rather than simply rename the temporary file into place, so as to preserve the target file's i-number. Not only is this gratuitously inefficient, but during the copy other clients cannot access the target file. Worse yet, a server crash will leave the file in an inconsistent state (though the client will restart the file transfer after the COMMIT fails).

A related problem occurs with file truncation. Applications often truncate files and then immediately overwrite them with similar versions. Thus, LBFS would benefit from having the previous contents of a truncated file when reconstructing the new contents. The obvious solution is to move truncated files into the trash directory and replace them with new, zero-length files. Unfortunately, the NFS interface will not let the server do this without changing the truncated file's i-number. To avoid losing the contents of trucated files, then, LBFS delays the deletion of temporary files after COMMITTMP RPCs. Thus, many truncated files will still have copies in the trash directory, and new versions can be reconstituted from those copies.

It is worth noting that the static i-number problem could be solved given a file system operation that truncates a file A to zero length and then atomically replaces the contents of a second file B with the previous contents of A. We can even afford to lose the original contents of A after an inopportune crash. In the case of COMMITTMP, the lost data will be resent by the client. Such a "truncate and update" operation would be efficient and easy to implement for most Unix physical file system layouts. It might in other situations serve as a more efficient alternative to the rename operation for atomically updating files. Unfortunately, the current LBFS server must make do without such an operation.

4.3 Client Implementation

The LBFS client uses the **xfs** device driver. **xfs** lets user-level programs implement a file system by passing messages to the kernel through a device node in `/dev`. We chose **xfs** for its suitability to whole-file caching. The driver notifies the LBFS client whenever it needs the contents of a file a user has opened, or whenever a file is closed and must be written back to the server. The LBFS client is responsible for fetching remote files and storing them in the local cache. It informs **xfs** of the bindings between files users have opened and files in the local cache. **xfs** then satisfies read and write requests directly from the cache, without the need to call into user-level code each time.

5 Evaluation

This section evaluates LBFS using several experiments. First, we examine the behavior of LBFS's content-based breakpoint chunking on static file sets. Next, we measure the bandwidth consumption and network utilization of LBFS under several common workloads and compare it to that of CIFS, NFS version 3 and AFS. Finally, we show that LBFS can improve end-to-end application performance when compared with AFS, CIFS, and NFS.

Our experiments were conducted on identical 1.4 GHz Athlon computers, each with 256 MBytes of RAM and a 7,200 RPM, 8.9 ms Seagate ST320414A IDE drive. The IDE drives are slower than common SCSI drives, which penalizes LBFS for performing more disk operations than other file systems. Except where otherwise noted, all file system clients ran on OpenBSD 2.9 and servers on FreeBSD 4.3. The AFS client was the version of ARLA bundled with BSD, configured with a 512 MByte cache. The AFS server was openafs 1.1.1 running on Linux 2.4.3. For the Microsoft Word experiments, we ran Office 2000 on a 900 MHz IBM ThinkPad T22 laptop with 256 MBytes of RAM, Windows 98, and openafs 1.1.1 with a 400 MByte cache.

The clients and servers in our experiments were connected by full-duplex 100 Mbit Ethernet through the Click [11] modular router, which can be configured to measure traffic and impose bandwidth limitations, delay, and loss. Click ran on a Linux 2.2.18 Athlon machine.

5.1 Repeated Data in Files

LBFS's content-based breakpoint chunking scheme reduces bandwidth only if different files or versions of the same file share common data. Fortunately, this occurs relatively frequently in practice. Table 1 summarizes the amount of commonality found between various files.

We examined emacs to see how much commonality there is between files under a software development workload. The emacs 20.7 source tree is 52.1 MBytes. However, if a client already has source for emacs 20.6 in its cache, it

Data	Given	Data size	New data	Overlap
emacs 20.7 source	emacs 20.6	52.1 MB	12.6 MB	76%
Build tree of emacs 20.7	—	20.2 MB	12.5 MB	38%
emacs 20.7 + printf executable	emacs 20.7	6.4 MB	2.9 MB	55%
emacs 20.7 executable	emacs 20.6	6.4 MB	5.1 MB	21%
Installation of emacs 20.7	emacs 20.6	43.8 MB	16.9 MB	61%
Elisp doc. + new page	original postscript	4.1 MB	0.4 MB	90%
MSWord doc. + edits	original MSWord	1.4 MB	0.4 MB	68%

Table 1: Amount of new data in a file or directory, given an older version.

only needs to download 12.6 MBytes to reconstitute the 20.7 source tree—a 76% savings. An emacs 20.7 build tree consumes 20.2 MBytes of disk space, but only contains 12.5 MBytes of unique chunks. Thus, writing a build tree, LBFS will save 38% even if the server starts with an empty chunk database. When adding a debugging printf to emacs 20.7 and recompiling, changing the size of the executable, the new binary has 55% in common with the old one. Between emacs-20.6 and 20.7, the two executables have 21% commonality. A full emacs 20.7 installation consumes 43.8 MBytes. However, 61% of this would not need to be transferred to a client that already had emacs 20.6 in its cache.

We also examined two document preparation workloads. When adding a page to the front of the emacs lisp manual, the new postscript file had 90% in common with the previous one. Unfortunately, if we added two pages, it changed the page numbering for more than the first chapter, and the commonality disappeared. From this we conclude that LBFS is suitable for postscript document previewing—for instance tuning a $\TeX$ document to get an equation to look right—but that between substantial revisions of a document there will be little commonality (unless the pages are numbered by chapter). We also used Microsoft Word to sprinkle references to LBFS in a paper about Windows 2000 disk performance, and found that the new version had 68% overlap with the original.

To investigate the behavior of LBFS's chunking algorithm, we ran *mkdb* on the server's `/usr/local` directory, using an 8 KByte chunk size and 48-byte moving window. `/usr/local` contained 354 MBytes of data in 10,702 files. *mkdb* broke the files into 42,466 chunks. 6% of the chunks appeared in 2 or more files. The generated database consumed 4.7 MBytes of space, or 1.3% the size of the directory. It took 9 minutes to generate the database. Figure 5 shows the distribution of chunk sizes. The median is 5.8K, and the mean 8,570 bytes, close to the expected value of 8,240 bytes. 11,379 breakpoints were suppressed by the 2K minimum chunk size requirement, while 75 breakpoints were inserted because of the 64K maximum chunk size limit. Note that the database does contain chunks shorter than 2K. These chunks come from files that are shorter than 2K and from the ends of larger files (since an end of file is always a chunk boundary).

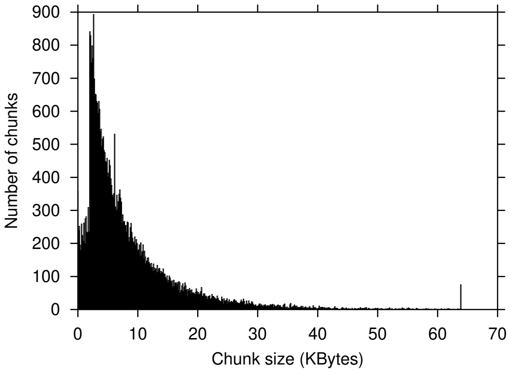

Figure 5: Distribution of chunk sizes in the `/usr/local` database. X-axis represents chunk sizes, in KBytes. Y-axis shows the number of chunks having that size.

Exp chunk size	% of data in shared chunks	
	24 B window	48 B window
2 KB	21.33%	21.30%
4 KB	19.29%	19.65%
8 KB	17.01%	18.01%

Table 2: Percentage of bytes in shared chunks in `/usr/local` for various chunk and window sizes. Minimum chunk size was always $1/4$ the expected chunk size.

Table 2 shows the amount of data in `/usr/local` that appears in shared chunks for various expected chunk sizes and breakpoint window sizes. As expected, smaller chunks yield somewhat greater commonality, as smaller common segments between files can be isolated. However, the increased cost of GETHASH and CONDWRITE traffic associated with smaller chunks outweighed the increased bandwidth savings in tests we performed. Window size does not appear to have a large effect on commonality.

5.2 Practical Workloads

We use three workloads to evaluate LBFS's ability to reduce bandwidth. In the first workload, MSWord, we open a 1.4 MByte Microsoft Word document, make the same ed-

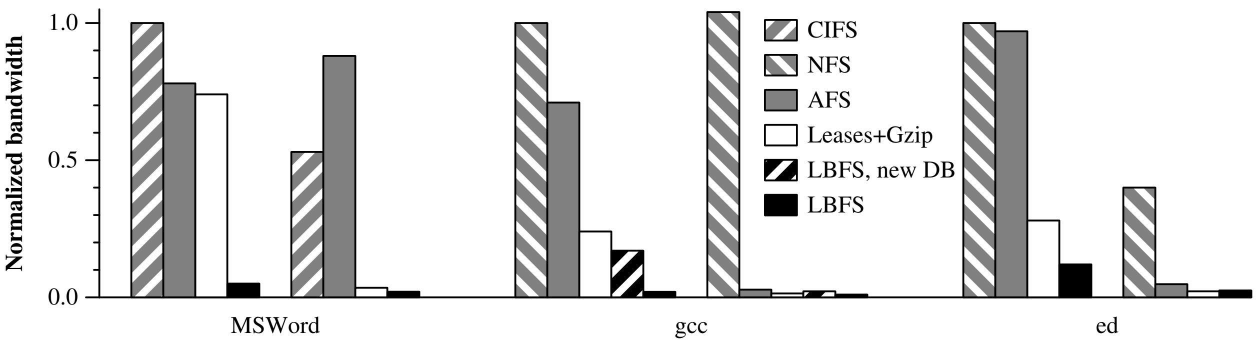

Figure 6: Normalized bandwidth consumed by three workloads. The first four bars of each workload show upstream bandwidth, the second four downstream bandwidth. The results are normalized against the upstream bandwidth of CIFS or NFS.

its as for Table 1, then measure the cost to save and close the file.[1] For the second workload, gcc, we simply recompile emacs 20.7 from source. The third workload, `ed`, involves making a series of changes to the perl 5.6.0 source tree to transform it into perl 5.6.1. Using the `expect` language, we scripted the `ed` text editor based on the output of `diff`. The benchmark saves files after text insertion operations if there have been 40 or more lines of editing. (For reference, emacs creates auto-save files every 300 characters typed.) These workloads reflect the common activities of document editing and software development.

We ran each benchmark over the client's native network file system—CIFS for Windows and NFS UDP for Unix.[2] We also ran all three benchmarks over AFS. To isolate the benefits of exploiting file commonality, we additionally measured a "Leases+Gzip" file system that uses LBFS's file caching, leases, and data compression, but not its chunking scheme. Finally, we ran the three workloads over LBFS.

For the MSWord benchmark, because LBFS only runs on Unix, we ran a Samba server on the OpenBSD LBFS client, re-exporting the LBFS file system with CIFS. We saved the Word document from the Windows 98 client to the Samba server over 100 Mbit Ethernet, and measured the traffic going through the Click router between the Samba server and the LBFS server. All MSWord experiments were conducted with a warm cache; the original files had been written on the same client through the file system under test. However, the LBFS server did not have output files from previous runs of the benchmark in its database.

For the gcc benchmark, the emacs sources had been unpacked through the file system under test. Emacs had also previously been compiled on the same file system. The intent was to to simulate what happens when one modifies a header file that requires an entire project to be recompiled. Though two successive compilations of emacs do not produce the same executable, there was substantial commonality between object files created by the benchmark and ones in the server's trash directory. To isolate this benefit, we also measured a compilation of emacs when the LBFS server was started with a new database not containing chunks from any previous compiles.

Because of a bug in `expect` on OpenBSD, we used a FreeBSD client for all instances of the `ed` benchmark. Like the MSWord benchmark, we ran `ed` with a warm client cache to a server that had not previously seen the output files. We also ran `ed` over an `ssh` remote login connection, to compare using a distributed file system to running a text editor remotely. To simulate some type-ahead, the benchmark sends one line at a time and waits for the line to echo.

5.3 Bandwidth Utilization

Figure 6 shows the bandwidth consumed by the client writing to and reading from the server under each of the three workloads. The bandwidth numbers are obtained from byte counters in the Click router. For this experiment, the router did not impose any delay, loss, or bandwidth limitations. (`ttcp` reported TCP throughput of 89 Mbit/sec between the client and server, and `ping` reported a round-trip time of 0.2 ms.) In each case, we separately report first the upstream traffic from client to server, then the downstream traffic from server to client. The numbers are normalized to the upstream (client to server) bandwidth of the native file system, CIFS on Windows and NFS on Unix.

Because AFS, Leases+Gzip, and LBFS all have large, on-disk caches, all three systems reduce the amount of downstream bandwidth from server to client when compared to the native file systems. For upstream bandwidth, the drops from CIFS and NFS bandwidth to AFS bandwidth represent savings gained from deferring writes to close time and eliding overwrites of the same data. The drops from AFS bandwidth to Leases+Gzip bandwidth represent savings from compression. Finally, the drops from Leases+Gzip bandwidth to LBFS bandwidth represent savings gained from the chunking scheme.

For the MSWord workload, the savings provided by the

[1] We did not enable Word's "fast saves" feature, as it neither reduced write bandwidth nor improved running time.

[2] NFS TCP performs worse than NFS UDP, probably because it has not been as extensively tested and tuned.

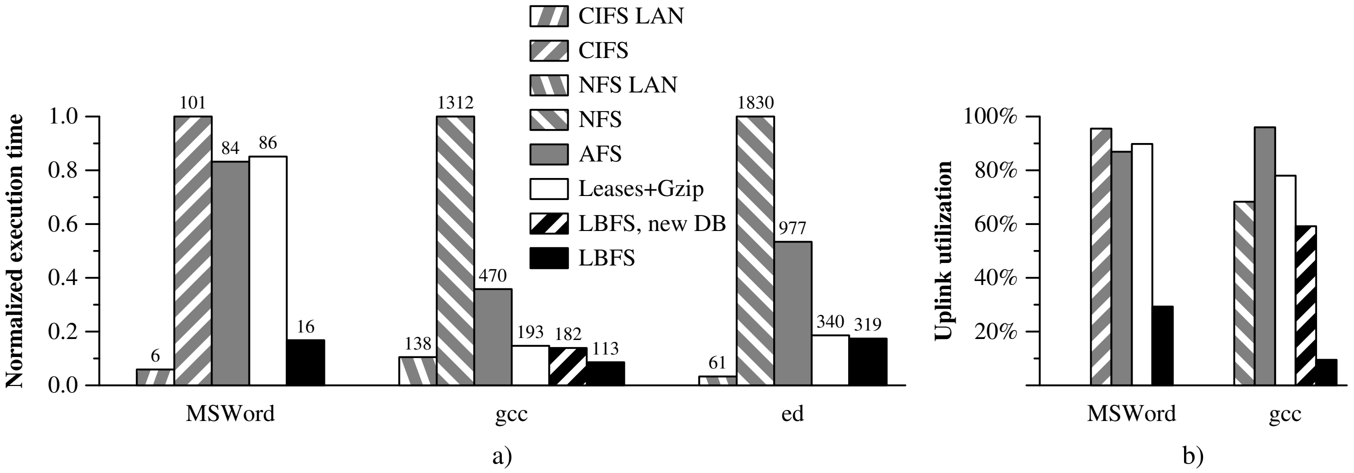

Figure 7: a) Normalized application performance on top of several file systems over a cable modem link with 384 Kbit/sec uplink and 1.5 Mbit/sec downlink. Execution times are normalized against CIFS or NFS results. Execution times in seconds appear on top of the bars. b) Uplink bandwidth utilization of the MSWord and gcc benchmarks.

chunking scheme come not only from commonality between the old and new versions of the document, but also from commonality with large temporary files that Word creates during saves. LBFS is able to reduce the upstream bandwidth by 15 times over Leases+Gzip, 16 times over AFS, and 20 times over CIFS. More careful analysis reveals that the Unix Samba server closes then reopens temporary files, requiring them to be transferred multiple times. These multiple transfers largely negate the benefits of gzip compression in Leases+Gzip. In contrast, LBFS exploits the files' common contents from one close to the next, consuming very little unnecessary traffic. AFS uses only slightly more bore bandwidth than Leases+Gzip, either because the extra closes are an artifact of the Unix Samba server, or perhaps because the Windows AFS implementation performs partial file caching.

For the gcc benchmark, the savings provided by the chunking scheme come from the fact that many of the compiled object files, libraries, and executables are similar or identical to files in the server's trash directory. Chunks only need to be written to the server where object files differ or files have been evicted from the trash directory. In this case, LBFS was able to reduce the upstream bandwidth by 15 times over Leases+Gzip, 46 times over AFS, and more than 64 times over NFS. Even without the benefit of old object files in the database, LBFS still reduces upstream bandwidth utilization because many object files, libraries, and executables share common data. When started with a new and empty chunk database, LBFS still used 30% less upstream bandwidth than Leases+Gzip.

In the `ed` case, the savings provided by the chunking scheme come from writing versions of files that share common chunks with older revisions. LBFS was able to reduce the upstream bandwidth by more than a factor of 2 over Leases+Gzip and 8 over AFS and NFS.

5.4 Application Performance

Figure 7a shows the normalized end-to-end application performance of the three workloads on a simulated cable modem link, with 1.5 Mbit/sec downstream bandwidth from server to client, 384 Kbit/sec upstream bandwidth from client to server, and 30 ms of round-trip latency. The execution times are normalized against CIFS or NFS results. For comparison, we also show the execution times of the native file system on a 100 Mbit/sec full-duplex LAN.

For the MSWord workload, LBFS was able to reduce the execution times from a potentially unusable 101 seconds with CIFS to a much more tolerable 16 seconds, more than 6 times faster. In fact, AFS takes 16 seconds to run the benchmark on a LAN, though CIFS takes only 6 seconds. The gcc workload took 113 seconds under LBFS with a populated database, 1.7 times faster than Leases+Gzip, 4 times faster than AFS, almost 12 times faster than NFS, and 18% faster than NFS on a LAN. With a new server database, LBFS still reduces the execution time by 6% over Leases+Gzip, though it is 32% slower than NFS on a LAN.

For both the MSWord and gcc workloads, Figure 7b shows that LBFS reduces network utilization, or the percentage of available bandwidth used by the file system. Over LBFS, gcc used only used only 9.5% of the 384 Kbit per second upstream link. In contrast, gcc under NFS used 68% and under AFS used 96%. For the MSWord benchmarks, LBFS was able to reduce the upstream network utilization from 87% and 96% with AFS and CIFS to 29%.

Figure 8 examines the effects of available network bandwidth on the performance of the gcc workload over LBFS, Leases+Gzip, and AFS. In these experiments, the simulated network has a fixed round trip time of 10 ms. This graph shows that LBFS is least affected by a reduction in available network bandwidth, because LBFS reduces the read and

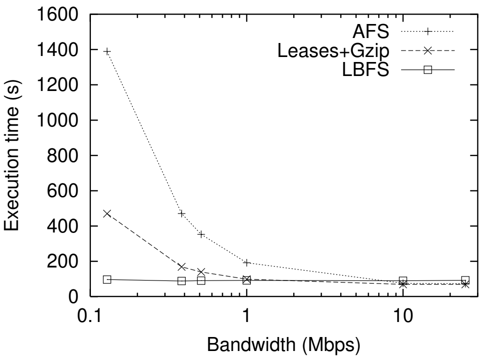

Figure 8: Performance of the gcc workload over various bandwidths with a fixed round-trip time of 10 ms.

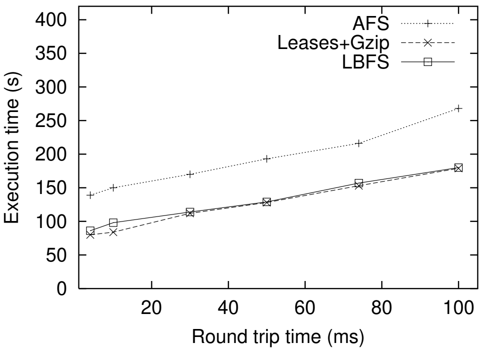

Figure 9: Performance of the gcc workload over a range of round-trip times with fixed 1.5 Mbit/sec symmetric links.

write bandwidth required by the workload to the point where CPU and network latency, not bandwidth, become the limiting factors.

Figure 9 examines the effects of network latency on LBFS, Leases+Gzip, and AFS performance. In these experiments, the simulated network has symmetric 1.5 Mbit per second links. Although the gcc workload uses more bandwidth over Leases+Gzip, the performance of the workload over LBFS and Leases+Gzip are roughly the same because the available network bandwidth is high enough. On the other hand, because gcc over AFS uses significantly more bandwidth, it performs worse than both LBFS and Leases+Gzip. This graph shows that the execution time of the gcc workload degrades similarly on all three file systems as latency increases.

Figure 7a also shows LBFS's performance on the `ed` benchmark, a 6% improvement over Leases+Gzip, 67% over AFS, and 83% over NFS. However, execution time is not the best measure of performance for interactive workloads. Users care about delays of over a second, but cannot differentiate much smaller ones that nonetheless affect the runtime of a scripted benchmark. Long delays are most often caused by TCP entering the backoff state. We therefore ran a shortened version of the `ed` benchmark over a network with

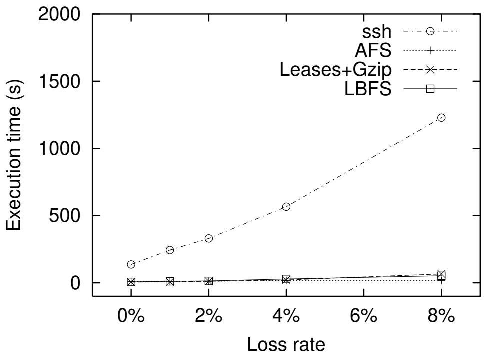

Figure 10: Performance of a shortened `ed` benchmark over various loss rates, on a network with fixed 1.5 Mbit/sec symmetric links and a fixed round-trip time of 10 ms.

simulated packet loss, comparing the performance of the network file systems with the `ssh` remote login program.

Figure 10 compares file systems to `ssh` under various loss rates. With no packet loss, `ssh` is slower than any file system, but the difference would not affect performance at the rate users type. However, as the loss rate increases, delays are imposed by TCP's backoff mechanism. As `ssh` never has more than a few packets in flight, every lost packet puts TCP into backoff, imposing a delay of one or more seconds while the user waits for typed characters to echo. The file systems outperform `ssh` for several reasons. First, LBFS and Leases+Gzip experience fewer losses by sending fewer total packets than `ssh`; the file systems both consume less bandwidth and send more data per packet. Second, when file systems transfer large files, TCP can get four or more packets in flight, allowing it to recover from a single loss with fast retransmission and avoid backoff. AFS uses UDP rather than TCP, and does not appear to reduce its sending rate as precipitously as TCP in the face of packet loss. We conclude that for the kind of editing in this benchmark, it is far preferable to use a network file system then to run an editor remotely.

6 Summary

LBFS is a network file system that saves bandwidth by taking advantage of commonality between files. LBFS breaks files into chunks based on contents, using the value of a hash function on small regions of the file to determine chunk boundaries. It indexes file chunks by their hash values, and subsequently looks up chunks to reconstruct files that contain the same data without sending that data over the network.

Under common operations such as editing documents and compiling software, LBFS can consume over an order of magnitude less bandwidth than traditional file systems. Such a dramatic savings in bandwidth makes LBFS practical for situations where other file systems cannot be used. In many situations, LBFS makes transparent remote file access a viable and less frustrating alternative to running interactive programs on remote machines.

7 Acknowledgments

We thank Chuck Blake, Frans Kaashoek, Butler Lampson, Robert Morris, Marc Waldman, and the anonymous reviewers for their feedback and suggestions. We thank Chuck Blake and Frank Dabek for help with hardware, Nickolai Zeldovich for setting up AFS, and Chuck Blake, Frank Dabek, and Bryan Ford for assistance with **xfs**.

References

[1] Brian Berliner. CVS II: Parellizing software development. In *Proceedings of the Winter 1990 USENIX Technical Conference*, Colorado Springs, CO, 1990.

[2] David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Westley Weimer, Christopher Wells, Ben Zhao, and John Kubiatowicz. Oceanstore: An exteremely wide-area storage system. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 190–201, Boston, MA, November 2000.

[3] Sergey Brin, James Davis, and Hector Garcia-Molina. Copy detection mechanisms for digital documents. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 398–409, San Jose, CA, May 1995.

[4] Andrei Broder. On the resemblance and containment of documents. *Compression and Complexity of Sequences*, pages 21–29, 1997.

[5] B. Callaghan, B. Pawlowski, and P. Staubach. NFS version 3 protocol specification. RFC 1813, Network Working Group, June 1995.

[6] FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, VA, April 1995.

[7] Cary G. Gray and David R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 202–210, Litchfield Park, AZ, December 1989.

[8] Björn Grönvall, Assar Westerlund, and Stephen Pink. The design of a multicast-based distributed file system. In *Proceedings of the Third Symposium on Operating System Design and Implementation*, pages 251–264, New Orleans, LA, February 1999.

[9] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[10] James J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[11] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(4):263–297, November 2000.

[12] Yui-Wah Lee, Kwong-Sak Leung, and M. Satyanarayanan. Operation-based update propagation in a mobile file system. In *Proceedings of the 1999 USENIX Technical Conference*, Monterey, CA, June 1999.

[13] Udi Manber. Finding similar files in a large file system. In *Proceedings of the Winter 1994 USENIX Technical Conference*, San Francisco, CA, January 1994.

[14] Timothy Mann, Andrew D. Birrell, Andy Hisgen, Chuck Jerian, and Garret Swart. A coherent distributed file cache with directory write-behind. *ACM Transactions on Computer Systems*, 12(2):123–164, May 1994.

[15] David Mazières. A toolkit for user-level file systems. In *Proceedings of the 2001 USENIX Technical Conference*, Boston, MA, June 2001.

[16] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 124–139, Kiawa Island, SC, 1999.

[17] Jeffrey C. Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta encoding and data compression for http. In *Proceedings of the 1997 ACM SIGCOMM Conference*, pages 181–194, Cannes, France, September 1997.

[18] Karin Petersen, Mike J. Spreitzer, and Douglas B. Terry. Flexible update propagation for weakly consistent replication. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 288–301, Saint-Malo, France, 1997.

[19] Michael O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[20] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. NFS version 4 protocol. RFC 3010, Network Working Group, December 2000.

[21] Neil T. Spring and David Wetherall. A protocol independent technique for eliminating redundant network traffic. In *Proceedings of the 2000 ACM SIGCOMM Conference*, pages 87–95, Stockholm, Sweden, August 2000.

[22] Walter F. Tichy. The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems*, 2(4):309–321, November 1984.

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

[24] Assar Westerlund and Johan Danielsson. Arla—a free AFS client. In *Proceedings of the 1998 USENIX, Freenix track*, New Orleans, LA, June 1998. USENIX.

[25] Haifeng Yu and Amin Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the 4rd Symposium on Operating Systems Design and Implementation*, San Diego, CA, 2000.