Appears in *Proceedings of the Fourth IEEE International Conference on Peer-to-Peer Computing* (IEEE P2P '04).

Improving Bandwidth Efficiency of Peer-to-Peer Storage

Patrick Eaton, Emil Ong, and John Kubiatowicz
Computer Science Division
University of California, Berkeley
Berkeley, California 94720–1776
Email: {eaton, emilong, kubi}@cs.berkeley.edu

Abstract

In this paper, we broaden the applicability of peer-to-peer storage infrastructures to weakly-connected clients. We present a client-side technique that exploits the commonality between consecutive versions of a file to reduce the bandwidth required to store and retrieve files in a peer-to-peer storage infrastructure. We then present a novel data structure that allows us to use this technique in an environment where peers cannot be trusted to perform operations over unencrypted data. We have implemented the technique in the OceanStore prototype. Additionally, with simulations, we have demonstrated that the technique can reduce client-perceived latency of write and read operations by up to 80% compared to techniques used in current systems.

1. Introduction

Peer-to-peer storage infrastructures promise to provide globally-accessible, highly-available storage. Most early implementations of such systems have been evaluated assuming clients are connected to the infrastructure with a high-bandwidth, low-latency connection characteristic of university, corporate, and laboratory environments [1, 3, 11]. But, there has already been a great amount of research and product development effort expended to build networked and distributed file systems for these environments, and such organizations are the very ones that possess the technological and financial resources to ensure the availability of high-performance storage systems maintained by full-time, professional staff. Furthermore, the measured performance (e.g. [3]) of peer-to-peer storage infrastructures indicates that they are not attractive replacements for systems currently in place.

These observations do not imply, however, that peer-to-peer storage infrastructures lack potential users. Indeed, we believe that such systems could empower a different class of users that have not previously had access to globally-accessible, highly-available storage. The class of users that we target are home and mobile users that access the network through low-bandwidth links.

Despite the promise of increased availability of broadband networking technologies, many residential and small-business users still depend on low-bandwidth connections. Worse, many technologies that are touted as "broadband"—such as cable modems, DSL modems, and wireless metropolitan networks—offer only a fraction of the bandwidth of technologies available to larger organizations. And for many rural customers, the future availability of even these "broadband" technologies is uncertain. Also, while network connectivity is increasingly available on a variety of mobile devices, these mobile devices—that could benefit so much from a highly-available storage infrastructure to prevent data loss due to theft, damage, and loss—often rely on congested public wireless connections or low-bandwidth cellular modems.

In this paper, we examine how to improve access to peer-to-peer storage infrastructures for this previously ignored class of users. Recognizing that home and mobile users are often limited by their low-bandwidth network connection, we focus on reducing the bandwidth required by users of peer-to-peer storage infrastructures. Certainly, the challenge of reducing the bandwidth required to communicate with a remote storage system to benefit weakly-connected clients has been well-studied. However, traditional approaches to this problem are not applicable in the peer-to-peer realm because they assume a trusted server that can manipulate user data in plaintext. The solution that we present is a client-side technique that recognizes commonality among consecutive versions of a file to reduce the bandwidth needed to store and retrieve files. It relies on a novel data structure that supports the efficient insertion and deletion of indivisible blocks of encrypted data.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 presents a client-side technique that exploits commonality among consecutive versions of a file to conserve bandwidth. Section 4 describes a data structure that allows us to use the technique in a peer-to-peer environment with limited trust. In Section 5, we detail our experimental implementation and present simulation results comparing the effectiveness of the proposed scheme to approaches used in current systems. Section 6 proposes future work and concludes.

2. Related Work

Our work has been inspired by a number of the peer-to-peer storage infrastructures presented in recent years. It also draws on research in other fields.

2.1. Peer-to-PeerStorage Infrastructures

A number of peer-to-peer storage systems have been proposed in recent years. While we cannot hope to review them all, we describe several notable systems and their salient features in this section.

Many peer-to-peer storage systems share a number of common features. Because individual machines in the infrastructure are not trusted to protect the integrity of user data, systems provide content addressable storage (CAS). In CAS systems, data is named by a secure hash of its content. Clients verify data retrieved from the infrastructure by comparing the hash of the data with the name by which the data was retrieved. To make large objects verifiable, a system can divide the object into multiple blocks and arrange the blocks in a tree [17]. Individual machines are also not trusted to protect the privacy of data. To protect against data theft, sensitive data must be appropriately encrypted.

In other respects, peer-to-peer storage systems differ significantly. Some infrastructures act as a large virtual disk or a distributed block store. In these systems, the client performs all computation locally, reading or writing blocks to the infrastructure when needed. Systems in this category include CFS [3] and PAST [13]. To write a file, a client divides the file into blocks and submits those blocks to the infrastructure for storage. To read a file, a client requests blocks from the infrastructure, interpreting them locally. Some systems choose not to divide the file, treating the whole file as a single block [13]; others divide a file into fix-sized blocks in a manner typical of a traditional file system [3].

Other systems provide infrastructural support for managing objects stored in the system. Systems in this category include FARSITE [1] and OceanStore [11]. These systems rely on groups of servers implementing Byzantine agreement protocols to manage stored objects. To modify an object, a client submits a signed update to the infrastructure. A Byzantine group receives the update, serializes it with other outstanding requests, cryptographically verifies client write permission, and applies the update. When a client reads an object, the group generates a signature attesting to the current version of the file; the actual data can then be retrieved from the CAS system. While the machines comprising the Byzantine group are trusted in aggregate to execute protocols, they are never trusted with the privacy of data.

In this paper, we assume an infrastructure like FARSITE or OceanStore that provides a point of serialization in the infrastructure responsible for committing updates and capable of verifying the most recent version of the data.

2.2. Other Related Work

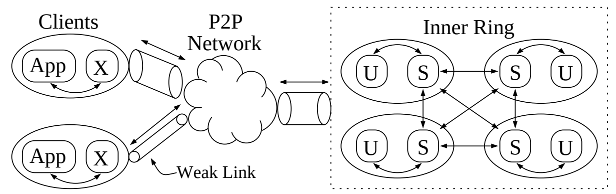
We also borrow techniques from other fields of research. Rabin [10] presented a computationally-efficient approach for generating unique fingerprints of data, used subsequently in several projects. Manber used the technique to find similar files in a large file system [6]. Spring and Wetherall used Rabin fingerprints to identity redundant network traffic [14]. And Muthitacharoen et. al. used the approach to detect commonality among consecutive versions of files to build the Low-Bandwidth File System (LBFS) [8]. Due to its impact on our own research, we discuss this last work further.

By identifying commonality among versions of the same file, LBFS reduced the amount of bandwidth used by clients communicating with a remote file server. Thus, LBFS was able to support clients connected to the file server through a low-bandwidth network connection. The technique assumed a trusted file server that could view and transform user data stored in plaintext. Depending on the workload, the technique was able to reduce execution time up to 90%.

Finally, the EXODUS Extensible Database System [2] developed a data structure to support efficiently the insertion and deletion of data at arbitrary offsets in an object. The data structure, a variation on a B^+-tree, used relative offsets to handle operations on bulk data. The algorithms, however, required that the database server be able to subdivide the data at arbitrary offsets.

3. Bandwidth-Efficient Peer-to-Peer Storage

In this section, we first describe our assumptions about the peer-to-peer storage infrastructure and then detail a client-side technique to reduce the bandwidth required to store or retrieve data in such an infrastructure.



Applications (App) on a client machine forward requests to a component (X) that transforms them for submission to the infrastructure. Inner ring machines contain components for serialization (S) and applying updates (U).

Figure 1.

3.1. Storage Infrastructure Assumptions

As described in Section 2, the research literature presents a wide variety of peer-to-peer storage systems. For our presentation, we assume a system like OceanStore [11] as shown in Figure 1.

The *inner ring* provides support for managing objects in the infrastructure. Each inner ring machine is logically divided into two components. The *serializer* collaborates with other inner ring machines to select a serial order over client update requests; this order is announced to the rest of the system via cryptographic certificates. To minimize the impact of malicious or faulty inner ring servers, the serializers can collaborate via a Byzantine agreement algorithm and threshold signatures. Once a serial order has been chosen, the serializer passes updates to the *updater* to transform data based on requests from clients. This process invests aggregate trust in the inner ring to perform updates and serialization correctly, but not individual trust in its members.

An inner ring manages a set of files or *data objects*. To modify a data object, a client submits a signed *update* to the inner ring responsible for that data object. An update is an ordered list of *actions* guarded by a *predicate*. The serializer orders the request and then passes it to the updater where the signature is verified to determine if the signing principal is permitted to modify the object. If the update is permitted, the updater evaluates the predicate and, if it is true, applies the actions to create a new version of the data object. The purpose of the predicate is usually to ensure consistency, as we will see in the next section. Finally, the serializer signs the result and sends to the client a response indicating whether the update succeeded or failed. After an update has been applied, the blocks that comprise the data object are named by a secure hash of their content and are thus immutable. The inner ring is then free to move blocks to other servers in the infrastructure.

To read a data object, the client must collect the component blocks. Because blocks are immutable and self-verifying, a client can retrieve the blocks from its local cache or from remote block servers, in addition to the inner ring responsible for the object. To ensure that it is collecting blocks from the current version of the data object, a client may request from the inner ring a signed certificate attesting to the current version.

A large fraction of the complexity of this system is in the serializer component. Consequently, we seek to avoid altering this mechanism. Instead, we focus on the data format, the predicates, and the actions to reduce bandwidth while preserving the privacy of user data.

3.2. Delta-Compressed Updates

Certainly, the literature contains previous work for reducing the bandwidth consumed between a client and a file server. Peer-to-peer storage systems, however, operate under a set of constraints which render previous techniques inappropriate. For example, previous techniques assume a trusted file server that can perform arbitrary transformations on data stored in plaintext. In peer-to-peer systems, however, because communications channels and other machines are untrusted, sensitive data must be encrypted to ensure privacy. Furthermore, because the data is encrypted, the infrastructure cannot transform the data in an arbitrary manner. Traditional schemes require that the file server be modified to participate in new protocols that often require multiple rounds of communication. In our model, the complexity of the serializer persuades us to search for techniques that retain the high-level request-response protocol used to communicate between the client and the infrastructure.

These constraints lead us to a client-side technique the uses delta-compression to reduce the size of the update that must be transmitted to the inner ring. The technique is designed for unmodified applications accessing arbitrary data.

Using delta-compression, we exploit the commonality that often exists between consecutive versions of a file. We compare a new version of a data object to its previous version, extract the changes made to the new version, and encode those changes in an update. Ideal compression would result in updates of minimum size. Though it does not provide ideal compression, the scheme presented here provides reasonable compression while remaining within the constraints of peer-to-peer systems and maintaining the storage and computational overheads of compression, encryption, and indexing to reasonable levels.

The compression technique relies on insertion and deletion operations that are not typically provided by file system APIs. To support this set of operations, the updater requires that data be stored in a specialized data format that will be discussed in more detail in Section 4.

File Summaries: At the core of our technique is the *file summary*. A file summary is a condensed representation of the contents of a data object. File summaries are similar in concept, if not in use, to file recipes used in the CASPER file system [15]. To create a file summary, a data object is first divided into smaller pieces, or *chunks*. An identifier is created for each chunk by computing a cryptographically-secure hash of the plaintext of the chunk's data. The file summary then records the hash along with the length and offset in the data object of each chunk.

To divide a file into chunks, we use Rabin fingerprints in a manner originally proposed in the Low-Bandwidth File System [8]. Rabin fingerprints identify chunk boundaries based on the content of a sliding window in the file and, consequently, tend to identify the same boundaries even as modifications may shift data offsets or change the size of the file. Rabin fingerprints divide a file into chunks such that the number of chunks that differ between consecutive versions of a file is roughly proportional to the amount of data modified between the versions. To avoid pathological cases that produce chunks that are unreasonably large or small, we define a minimum and maximum chunk size. Boundaries are ignored if the resulting chunk would be too small; a boundary is inserted if the chunk reaches the maximum size.

Updates from File Summaries: As the data stored in the data object changes, so too does the object's file summary. Rather than compute a new file summary with every write operation, a client adopts the write-on-close approach used in AFS [4]. A client creates a new file summary on every file close operation, if the file was originally opened for writing. The client uses the file summary of a modified file to construct an update to send to the inner ring. To exploit the commonality that is typical between consecutive versions of the same file, the client compares the newly created file summary to the file summary of the previous version. The result of the comparison is combined with chunks containing new or modified data to create an update which is forwarded to the inner ring.

This procedure tends to create efficient updates; that is, the size of the update is roughly equal to the amount of data that actually changed. It does this by translating file system operations into a more flexible set of operations. Unmodified applications still use the standard file system interface with truncate, append, and overwrite operations. With only these operations, however, applications are often forced to overwrite the tail of a file, even if only a small amount of data is changed. A naive translation of file system operations to updates would also result in overwriting the tail of a data object. The proposed translation process recasts the modification as a series of insert and delete operations that more succinctly describe the changes made to the file. We present a data structure that enables the inner ring to support those transformations in Section 4.

Note that fingerprinting and creation of file summaries must be performed over the plaintext of the data. If the data were to be encrypted (using the standard cipher block chaining mode) before it was divided into chunks, later chunks would depend upon earlier chunks. This would eliminate the possibility of detecting file commonality after the first modification point in the file and consequently increase the size of the updates created by the client.

Update Predicates: The inner ring will create the new version of the data object intended by the client only if it applies the update to the same version of the data object used in the file summary comparison at the client. Applying the update against a different version could corrupt data. To ensure data object consistency, each update is predicated on the condition that the current version of the data object managed at the inner ring is the same version used to construct the update. If the condition does not hold when the inner ring processes the update, the client will receive an update failure response. The client can then elect to fetch the new version of the data object and construct a new update or overwrite the entire file.

We take an optimistic approach to file consistency, constructing and submitting updates based on the current version of the file at client machine without contacting the inner ring. We believe an optimistic approach is appropriate because we expect it to be uncommon for clients accessing peer-to-peer storage through a weak network connection to be interactively sharing files in the system.

File Summaries as Soft State: After committing an update to the infrastructure, a file summary can be considered soft state. In the common case, the client should cache the file summary for use during the next write operation. However, a client may discard a file summary at any time to pare the amount of state that it must record. It can simply recreate the file summary the next time an application opens a file for writing. Certainly, this approach incurs a significant computation cost, but for many clients, computational capacity is abundant compared to scarce network bandwidth. Alternatively, the client may choose to neglect altogether the file summary of the previous version, instead sending the entire file in the next update. This approach may be appropriate for small files.

3.3. Caching for Read Performance

The approach described in Section 3.2 reduces the client-perceived latency to commit updates at the inner ring by decreasing the amount of data that must be transferred across the network. Combining that technique with aggressive client-side caching further improves service for weakly-connected clients.

Most obviously, caching can reduce the latency to read data, even if the cache contains stale data. If a client has a previous version of a data object cached, it only needs to retrieve chunks newer than the cached version to reconstruct the current version. Because the process that creates the updates tends to preserve unmodified chunks, a client with a recent version of a data object cached needs to retrieve a number of new chunks proportional to the amount of data that has actually changed.

Caching can also help clients tolerate low-bandwidth connectivity by providing them a view of data that reflects their updates, even before they have been certified by the inner ring. Caching, then, allows clients to work at the speed of the local machine while pipelining updates to the inner ring as quickly as the network will support. We believe this to be an important optimization for future study.

4. A Data Structure for Efficient Updates

The technique presented in the previous section assumes that the inner ring can apply the update created by the client. Unfortunately, this assumption is not trivially true. Complicating the issue are the restricted trust assumptions inherent in a peer-to-peer system and the operations used in the update. These issues place requirements on how the infrastructure represents user data. In this section, we enumerate the requirements of this representation and present a data structure that meets these requirements. We also present a simple extension that allows a storage infrastructure to support simultaneously legacy applications using the approach described previously while providing additional flexibility and efficiency for specialized applications.

4.1. Requirements

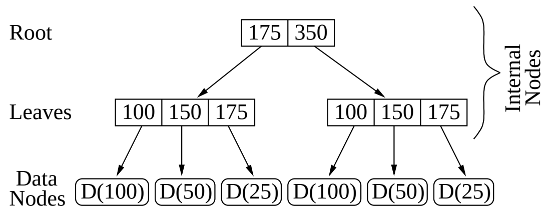
The file summaries and updates reference chunks of varying sizes, as identified by Rabin fingerprints. The representation, consequently, needs to index efficiently blocks of unequal size. This requirement renders inode-like schemes inappropriate.

To apply the updates, the representation must support flexible transformation operations. In addition to truncate and append, the representation must also support insert and delete operations. Furthermore, these operations need to be incremental. That is, if only a small portion of the data is modified, a correspondingly small part of the data structure should change.

Finally, the trust model of the peer-to-peer environment further restricts the representation. Because data privacy cannot be ensured, clients must encrypt sensitive data. Where previous approaches have assumed that blocks of data could be repartitioned at the server [2], the representation must treat chunks as opaque and indivisible.

4.2. Solution

In this section, we present a data structure that meets the requirements listed above. The data structure presented here



This tree maintains an index over six blocks of data, shown at the leaves. The root shows that the first 175 bytes of data can be found via the left subtree; data between offset 175 and 350 can be accessed via the right subtree. The index records *relative* offsets—note, that the two leaf nodes contain the same counts.

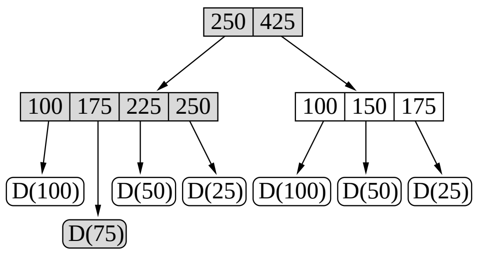
Figure 2.

supports insert and delete operations at block boundaries. This is sufficient to apply the updates created using the technique of Section 3. An extension that allows arbitrary insert and delete operations is given in the next section.

Our solution was inspired by the large-object data structure in the EXODUS database system [2]. The data structure, a variation of the B^+-tree, is a balanced tree with all data stored at the leaves. We use the term balanced to mean that the number of children is balanced, not the amount of data in each child.

The tree is composed of two types of nodes. At the base of the tree are *data nodes*. A data node contains an uninterpreted block of potentially encrypted user data. If the data is encrypted, the data node will also contain the initialization vector required for decrypting the data. Encrypted data must not rely on data in adjacent nodes; thus, if cipher block chaining is used, the chaining must terminate in each data node. All other information needed to decrypt the data, such as the encryption algorithm used, is stored in the data object metadata, not in each data node.

Other nodes, called *internal nodes*, combine to form an index over the data nodes. The top-most internal node is called the *root*; internal nodes that point directly to data nodes are called *leaves*. Internal nodes store up to $2n + 1$ (*count*, *pointer*) tuples, where n is the *degree* of the tree. All internal nodes except the root store between n and $2n + 1$ non-null tuples at all times; the root may contain fewer tuples. The *count* field is an integer that records the number of bytes accessible from the current node via children referenced from the given tuple and all other tuples to the left of the current tuple. By convention, $count[-1] = 0$. The key feature of this structure is that the *count* field records the *relative*, rather than *absolute*, offset of data within the structure. The *pointer* field is a secure hash that identifies and verifies the child node. The *pointer* fields form a Merkle [7] tree over the structure, making it self-verifiable. Figure 2 shows an example of this data structure.



Maintaining relative offsets enables efficient incremental insert operations. Shown is the result of inserting 75 bytes at offset 100 in the data structure shown in Figure 2. Blocks modified during the operation are shaded.

Figure 3.

We now describe the algorithm used to insert a block of ciphertext in the data structure. While space constraints preclude us from defining algorithms for all operations, other operations are implemented using the same set of basic techniques.

The insert operation allows a client to insert a chunk in an object. Let *Data* be the block of potentially encrypted text to be inserted, and let *InsertPoint* be the offset at which to insert the data. Recall that the insertion point must lie on an existing block boundary for the current discussion. Throughout the discussion, refer to Figure 3 which illustrates the result of an insert operation on the data object of Figure 2.

1. The algorithm must not descend into a node that cannot accommodate a new child. If the root is full, create a new internal node to be the root, assign the old root to be its only child, and split that child. The split procedure works in a fashion similar to the traditional B-tree split operation. Let $Node \leftarrow root$. Proceed to Step 2.
2. *Node* contains an array of tuples describing the range of data that may be reached from each child node. Using this state, find the child that is the parent of the insertion point; that is find *i* such that $count(i-1) < InsertPoint <= count(i)$. Let $count(j) \leftarrow count(j) + Data.size$ for all $j >= i$. Also, mark the child at $pointer(i)$ as modified. If *Node* is a leaf node, skip to Step 4; else proceed to Step 3.
3. Prepare to descend to the child. If the child referenced by *pointer(i)* is full, split that child and adjust *i* accordingly to point to the new child that is the parent of the insertion point. Let $Node \leftarrow pointer(i)$. To handle the relative offsets, let $InsertPoint \leftarrow InsertPoint - count(i-1)$. Recurse to Step 2.
4. Insert a new tuple at offset *i* with $count(i) = count(i-1) + Data.size$ and $pointer(i) = \text{HASH}(Data)$. The operation is now complete.

While the new data has been inserted, the ability to verify the data structure has been lost. To restore the verifiability, after all actions of the update have been applied, perform a depth-first traversal of the data structure, recomputing the secure hash of all modified blocks and storing the result in the parent. The traversal need not descend into any children that were not modified. Thus the amount of data that must be hashed after any update is proportional to the amount of modified. The algorithm could be extended to maintain verifiability after each modification. However, most updates contain multiple operations, and we can save computation by restoring verifiability only after all modifications have been completed.

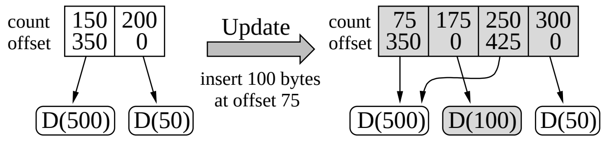
4.3. Supporting Arbitrary Insertions

The solution presented in the previous section supports points of modification only at existing block boundaries. While this functionality is sufficient to support the technique presented in Section 3, a simple extension can enable even more flexible transformations.

The extension allows the data structure to support insertion and deletion at *arbitrary* offsets. In all leaf internal nodes, we add to the tuples an additional field called *offset*. The added field is used to point into the middle of a block of encrypted data stored in the data node. The extension is illustrated in Figure 4. In the example, before the update, the data structure holds two chunks of data, one of 500 bytes and one of 50 bytes. The first $count[0] = 150$ bytes can be found starting at offset $offset[0] = 350$ of the encrypted data stored in the first data node. The first 350 bytes in that block could referenced by a tuple that is not shown or could be unreferenced in the current version. The figure also shows the result of updating the data object. After inserting 100 bytes at offset 75, the first 75 bytes of data can be found starting at offset 350 of the first data node. The bytes in the range 175 to 250 can also be found in the same data node, starting at offset 425. The algorithms to modify this data structure are similar to those used to modify the data structure in Section 4.2.

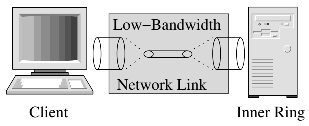
While this extension does provide additional capabilities, it comes with additional complications, especially related to security and performance. For example, the data nodes may store data that has been deleted from the current version of the data object. By manipulating the blocks directly, a malicious attacker could reconstruct information about the change history of the file. Of course, the attacker would still need a key to decrypt the data. This dead data is also a performance liability. Retrieving the data object from the infrastructure may require retrieving extra data that is no longer relevant to the current version of the document.

Despite the complications, the extension opens several possibilities. For example, with a carefully-defined data for-



A simple extension allows the data structure to support insertion and deletion of data at *arbitrary* offsets.

Figure 4.



Our evaluation uses a simple simulator with an emulated low-bandwidth network link.

Figure 5.

mat, an application with a stale copy of the data object could submit an update without retrieving the current version and merging the changes. Alternatively, an application could dynamically trade-off computation and bandwidth depending on the current level of connectivity. In low-bandwidth conditions, it could submit the smallest update possible. When connectivity is restored, the application could retrieve the data object, defragment the data, and rewrite it to the infrastructure in the more efficient manner. Finally, an application could be written to exploit the asymmetric nature of many networking technologies. For example, a typical cable modem provides three times the amount of downstream bandwidth as upstream bandwidth. An application could submit small updates that can be uploaded to the infrastructure quickly even if they result in less efficient storage in the data structure knowing that extra downstream bandwidth will enable quick retrieval of the document. We are exploring how applications could use techniques like these to improve further client-perceived performance.

5. Implementation and Evaluation

To validate the approach described in Sections 3 and 4, we have implemented the scheme in the OceanStore prototype [11]. We chose not to evaluate the approach based on this implementation because of the performance variability of some of the components of the full design.

Instead, to evaluate the technique, we created an abstract model of a peer-to-peer storage infrastructure. By modeling the infrastructure, we were able to isolate and control features relevant to the evaluation while ignoring other issues that could obscure the results. One key abstraction is that we model the inner ring as a single server, eliminating the performance variability caused by Byzantine agreement. This abstraction has a conservative effect on results. In a full system, the client would send a request to a single inner ring member, who would then forward the request to the other inner ring members. We assume, however, that the bandwidth between inner ring machines will be much greater than that of the weak connection between a client and the infrastructure. Thus the system bottleneck is in the client link, so we are able to observe most of the benefit by simulating only that connection. Another key abstraction is that the client and inner ring communicate via a direct point-to-point connection to eliminate the variable latency of routing messages through a peer-to-peer networking layer.

We then developed a simple simulator to model the abstraction. The simulator is written in Java using the event-driven architecture toolkit developed as part of the Bamboo project [12]. A client simulator executes traces of file activity, computes file summaries and updates, and submits requests to the inner ring. An inner ring simulator serializes and applies updates and serves blocks for read requests. The two simulators communicate using TCP through an emulated restricted network connection. This configuration is shown in Figure 5.

For comparison, we implemented two alternative approaches used in current systems. The *Whole* approach treats the entire file as a single chunk. If any single byte of the file is changed, all data must be shipped to the inner ring for update. This is analogous to the approach employed in PAST [13]. The *Block* approach divides a file in to fix-sized blocks, much as a traditional file system divides a file into blocks for storage on disk. In this scheme, the inner ring represents the data as a standard B$^+$-tree. When a file is modified, only those blocks that have changed need to be submitted to the inner ring. In simulations, each block is 4096 bytes. This is similar to the approach used in CFS [3].

We also compare the experimental results against a computed ideal latency. The *Ideal* latency is a computed quantity assuming that operation latency depends only on the speed of network transmission. The size of the update is computed using a binary diff algorithm based on a combination of techniques used in rsync [16] and Xdelta [5] and assumes that the inner ring can perform arbitrary transformations on the data. Furthermore, it is assumed that computation is infinitely fast and data can be stored as ciphertext with no storage overhead.

In the *FileSummary* approach, chunks are identified using Rabin fingerprints with a minimum chunk size of 4096 bytes and a maximum chunk size of 16,384 bytes. We use the SHA-1 algorithm for secure hashing and the algorithm best known for its implementation in the UNIX diff utility [9] for comparing file summaries. All user data is encrypted using the Rijndael/AES cipher with 128-bit keys.

Technology	Latency (ms)	Bandwidth (kb/s)
Cellular modem	100	10
Telephone Modem	100	56
Cable Modem	30	384 down / 128 up
LAN Connection	10	1000

Table 1. The parameters that control the weak link between the client and inner ring.

5.1. Experimental Setup

The two components of the simulator both run simultaneously on a single Dell Dimension 8300 desktop with one 3 GHz Pentium 4 processor and 1 GB of RAM running Gentoo Linux 1.4.

To emulate low-bandwidth network connections, we impose a delivery delay on all messages at the receiver. We assume a simple model in which the bandwidth and latency are well-defined and constant between the client and server. We compute the delay using the familiar $\alpha + \beta \cdot n$ formula, where α corresponds to latency, β corresponds to bandwidth, and n is the size of the current message in bytes. Since the actual network connection between the two components is the loopback network device, we ignore any latency it imposes.

Table 1 shows the bandwidth and latency parameters used. They correspond roughly to a cellular modem, a telephone modem, a cable modem, and a LAN connection.

The presented performance numbers are the mean of three trials. The variance was negligible in all cases.

5.2. Workloads

We evaluate our scheme using three workloads summarized in Table 2. The first workload is composed of consecutive versions of a document created using Microsoft Word 2000. The 11-page document outlines the engineering results of a public works design project; it includes several figures and graphs. To create the second version, we performed a global search and replace operation to change the name of the city contracting the project. This changed 15 instances covering most pages of the document. This will be called the *Word* workload.

The second workload is consecutive source code releases from February 9 and 13, 2004 of the Bamboo DHT and event-driven programming toolkit [1]. The February 13th release modified 26 files ranging in size between 1.5 kB and 70 kB. This will be called the *Bamboo* workload.

1 Current and previous Bamboo releases are available at http://www.bamboo-dht.org/.

Name	Files	Total	Commonality		
		Size (kB)	Block	Rabin	Ideal
Word	1	700	0.6%	82.4%	98.7%
Bamboo	26	435	10.9%	20.4%	96.8%
Mbox	1	80-100	81.8%	78.4%	82.1%

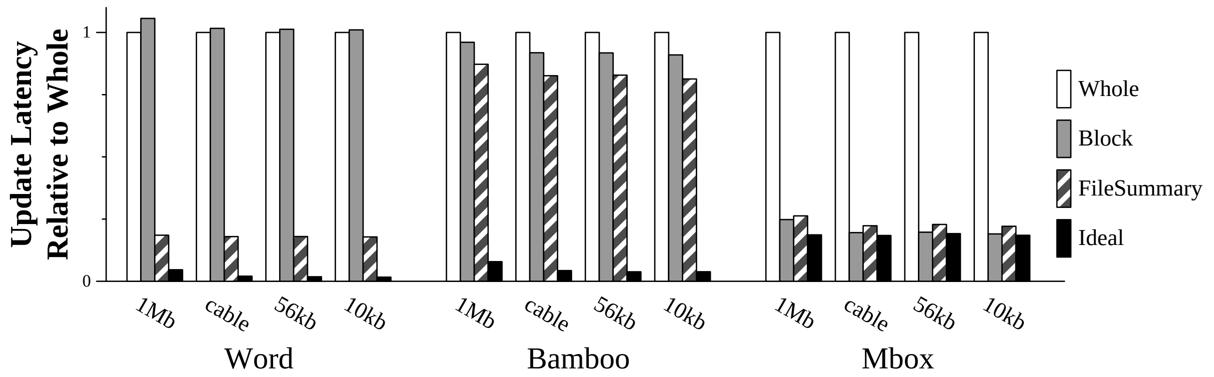
Table 2. The workloads used to evaluate the effectiveness of the system varied in size and in commonality between consecutive versions using the different chunking strategies.

The final workload uses several versions of a user's email folder stored in mbox format. The user saves 5 new messages to the folder increasing the size of the folder by 25%. In the mbox format, messages are always appended to the end of a mail folder, leaving the head of the file the same. This workload will be referred to as the *Mbox* workload.

Table 2 also shows the percentage of commonality found in the workloads using different chunking techniques. We define commonality to be the ratio of bytes found in chunks from the previous version to the total bytes in the new version. Of course, the Whole approach (not shown in table) is unable to identify any commonality in any workload. The Block approach is able to detect very little commonality in the Word workload because the application modifies the header on every update. On the Bamboo workload, it can recognize commonality that occurs before the insertion point in some files. Because the Mbox workload appends data to the end of the file, it can successfully recognize nearly all of the common data from the previous version. The FileSummary approach is able to detect some commonality in all workloads. On the Word workload, it recognizes that most of the data can be found in the previous version, despite the change in the header. Similarly, on the Bamboo workload, it is able to extract additional commonality after the point of modification that the Block approach cannot. It detects a similar amount of common data in the Mbox workload. The slight variation is due to the difference in chunk boundaries between the techniques. The Ideal computation is able to detect a large amount of commonality in all workloads. It is able to extract significantly more commonality because it assumes the inner ring can perform arbitrary transformations and it can define transformations on the byte, rather than the chunk, granularity.

5.3. Experimental Results

We begin our evaluation by measuring the costs incurred by the client while using the FileSummary scheme. One such cost is the additional storage required to store file summaries. The overhead of storing file summaries for the three



The impact on client-perceived write latency of the different workloads across a variety of networking technologies.

Figure 6.

studied workloads ranges from 4.3 to 5.1 megabytes per gigabyte of stored data. A client must also compute chunk boundaries and translate file summaries into updates. Using unoptimized Java code, the client can detect chunks using Rabin fingerprints at a rate of 200 ms per megabyte of data. The cost of creating the update is dependent on the length of the file and the number of new chunks. On the Word workload, for example, it takes 120 ms to create the update; of this time, however, about 50 ms is spent on the unavoidable task of encrypting the data.

Next, we measure the impact of our technique on write latency. Figure 6 shows the client-perceived latency of writing the various workloads to the server. We measure the time to update the data object(s) stored on the inner ring to the second version in the workload.

The FileSummary approach performs very well on the Word and Mbox workloads. In the Word workload, it vastly outperforms the other variations by recognizing a larger amount of commonality between versions. In the Mbox workload, where the Block approach is able to recognize a similar amount of commonality, the FileSummary scheme performs roughly equal to the simpler, but less flexible Block scheme. The FileSummary approach is slightly slower because the Block scheme recognizes more common data (see Table 2) and has a lower computational overhead. Even on this workload tailored to benefit the Block scheme, however, the FileSummary approach performs competitively.

While the FileSummary approach also performs better than other approaches on the Bamboo workload, the overhead compared to the Ideal scheme is much higher. The Bamboo workload has many small changes spread throughout the files. The slow performance of the FileSummary scheme relative to Ideal represents the cost of ensuring privacy and efficiency in a peer-to-peer system—the FileSummary approach must modify whole chunks rather than individual bytes because data encryption restricts the transformations that can be performed by the inner ring and the ciphertext must be managed in larger blocks for efficiency.

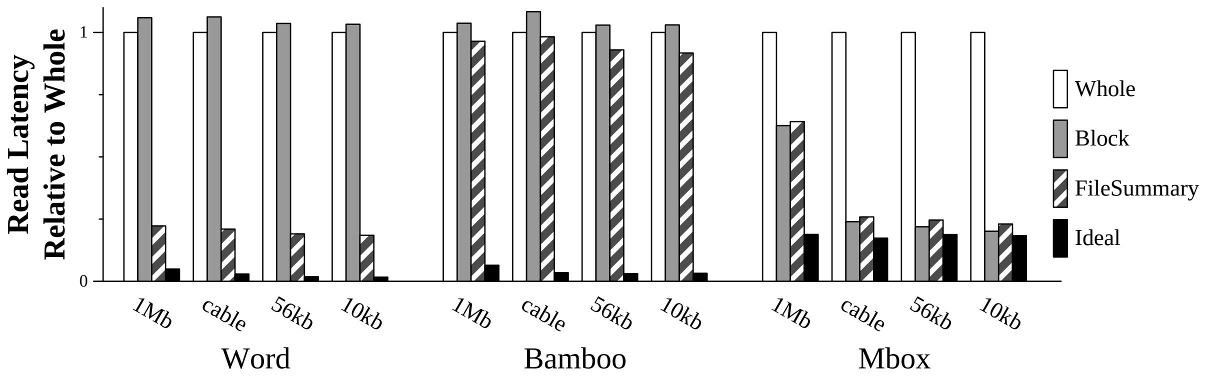
We also measure the impact of the technique on read latency. We assume that the client has a copy of the first version of the workload in the local cache. Upon requesting the file, the client discovers that a newer version has been created and that it must retrieve chunks to reconstruct the new version locally. This situation would arise for a user that accesses data from multiple machines or devices.

Figure 7 presents the results of this test. The FileSummary scheme is more than 80% faster on Word workload. On the Mbox workload, the Block scheme performs slightly better than the FileSummary scheme because the Block scheme detects slightly more commonality among files. On the Bamboo workload the FileSummary scheme is almost 10% faster. Note that when reading data from the infrastructure, the Block and FileSummary schemes pay a penalty for retrieving the internal nodes of the data structure. The Block approach actually performs worse than the Whole method because of this overhead coupled with the lack of detected commonality. The FileSummary scheme, however, is able to detect sufficient commonality to overcome this penalty and perform better than the Whole approach.

6. Conclusion and Future Work

Peer-to-peer storage infrastructures promise to provide globally-accessible, highly-available storage users. These infrastructures could be of great benefit for home and mobile users who do not currently have access to storage with such properties. A defining characteristic of this class of users is network connectivity through low-bandwidth links.

We presented a client-side technique that exploits the commonality between consecutive versions of files to reduce bandwidth required to store and retrieve objects and developed a novel data structure that supports efficient insertion and deletion of ciphertext blocks of variable size



The impact on client-perceived read latency of the different workloads across a variety of networking technologies.

Figure 7.

to support the technique. Both the technique and the data structure integrate smoothly into existing systems without revealing user data to the infrastructure.

We implemented the scheme in the OceanStore prototype. We also created a simulator to understand the performance characteristics of the technique. Simulations show that the approach reduces read and write latency up to more than 80% compared to techniques used in current systems.

In the future, we hope to measure the performance improvement over a wider range of real-world workloads. Finally, we would like to explore the potential benefits of using the extended features of the data structure.

Acknowledgments

The authors would like to thank Steven Czerwinski and the anonymous reviewers for comments on drafts of this paper. We would also like to thank the OceanStore group for their development of the Pond prototype used in this work.

References

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. of USENIX OSDI*, pages 1–14, Dec. 2002.

[2] M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita. Object and file management in the EXODUS extensible database system. In *Proc. of VLDB*, pages 91–100, Aug. 1986.

[3] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of ACM SOSP*, pages 202–215, Oct. 2001.

[4] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, Feb. 1988.

[5] J. P. MacDonald. File system support for delta compression. Master's thesis, University of California, Berkeley, 2000.

[6] U. Manber. Finding similar files in a large file system. In *Proc. of USENIX Winter Technical Conference*, pages 1–10, Jan. 1994.

[7] R. C. Merkle. A digital signature based on a conventional encryption function. In *Proc. of CRYPTO*, pages 369–378, 1988.

[8] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proc. of ACM SOSP*, pages 174–187, Chateau Lake Louise, Banff, Canada, Oct. 2001.

[9] E. Myers. An O(ND) difference algorithm and its variations. In *Algorithmica*, pages 251–266, 1986.

[10] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Department of Computer Science, Harvard University, 1981.

[11] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore prototype. In *Proc. of USENIX FAST*, pages 1–14, Mar. 2003.

[12] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a dht. In *Proc. of USENIX Technical Conference*, June 2004.

[13] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of ACM SOSP*, pages 188–201, Oct. 2001.

[14] N. T. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proc. of ACM SIGCOMM*, pages 87–95, Aug. 2000.

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

[16] A. Tridgell and P. Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, Department of Computer Science, Australian National University, 1996.

[17] H. Weatherspoon, C. Wells, and J. Kubiatowicz. Naming and integrity: Self-verifying data in peer-to-peer systems. In *Proc of FuDiCo*, pages 91–94, June 2002.