Wide-area cooperative storage with CFS

Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, Ion Stoica*
MIT Laboratory for Computer Science
chord@lcs.mit.edu
http://pdos.lcs.mit.edu/chord/

Abstract

The Cooperative File System (CFS) is a new peer-to-peer read-only storage system that provides provable guarantees for the efficiency, robustness, and load-balance of file storage and retrieval. CFS does this with a completely decentralized architecture that can scale to large systems. CFS servers provide a distributed hash table (DHash) for block storage. CFS clients interpret DHash blocks as a file system. DHash distributes and caches blocks at a fine granularity to achieve load balance, uses replication for robustness, and decreases latency with server selection. DHash finds blocks using the Chord location protocol, which operates in time logarithmic in the number of servers.

CFS is implemented using the SFS file system toolkit and runs on Linux, OpenBSD, and FreeBSD. Experience on a globally deployed prototype shows that CFS delivers data to clients as fast as FTP. Controlled tests show that CFS is scalable: with 4,096 servers, looking up a block of data involves contacting only seven servers. The tests also demonstrate nearly perfect robustness and unimpaired performance even when as many as half the servers fail.

1. Introduction

Existing peer-to-peer systems (such as Napster [20], Gnutella [11], and Freenet [6]) demonstrate the benefits of cooperative storage and serving: fault tolerance, load balance, and the ability to harness idle storage and network resources. Accompanying these benefits are a number of design challenges. A peer-to-peer architecture should be symmetric and decentralized, and should operate well with unmanaged volunteer participants. Finding desired data in a large system must be fast; servers must be able to join and leave the system frequently without affecting its robustness or efficiency; and load must be balanced across the available servers. While the peer-to-peer systems in common use solve some of these problems, none solves all of them. CFS (the Cooperative File System) is a new design that meets all of these challenges.

A CFS file system exists as a set of blocks distributed over the available CFS servers. CFS client software interprets the stored blocks as file system data and meta-data and presents an ordinary read-only file-system interface to applications. The core of the CFS software consists of two layers, DHash and Chord. The DHash (distributed hash) layer performs block fetches for the client, distributes the blocks among the servers, and maintains cached and replicated copies. DHash uses the Chord [31] distributed lookup system to locate the servers responsible for a block. This table summarizes the CFS software layering:

Layer	Responsibility
`FS`	Interprets blocks as files; presents a file system interface to applications.
`DHash`	Stores unstructured data blocks reliably.
`Chord`	Maintains routing tables used to find blocks.

Chord implements a hash-like operation that maps from block identifiers to servers. Chord assigns each server an identifier drawn from the same 160-bit identifier space as block identifiers. These identifiers can be thought of as points on a circle. The mapping that Chord implements takes a block's ID and yields the block's *successor*, the server whose ID most closely follows the block's ID on the identifier circle. To implement this mapping, Chord maintains at each server a table with information about $O(\log N)$ other servers, where N is the total number of servers. A Chord lookup sends messages to $O(\log N)$ servers to consult their tables. Thus CFS can find data efficiently even with a large number of servers, and servers can join and leave the system with few table updates.

DHash layers block management on top of Chord. DHash provides load balance for popular large files by arranging to spread the blocks of each file over many servers. To balance the load imposed by popular small files, DHash caches each block at servers likely to be consulted by future Chord lookups for that block. DHash supports pre-fetching to decrease download latency. DHash replicates each block at a small number of servers, to provide fault tolerance. DHash enforces weak quotas on the amount of data each server can inject, to deter abuse. Finally, DHash allows control over the number of *virtual servers* per server, to provide control over how much data a server must store on behalf of others.

CFS has been implemented using the SFS toolkit [16]. This paper reports experimental results from a small international deployment of CFS servers and from a large-scale controlled test-bed. These results confirm the contributions of the CFS design:

- an aggressive approach to load balance by spreading file blocks randomly over servers;

* University of California, Berkeley

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Space and Naval Warfare Systems Center, San Diego, under contract N66001-00-1-8933.

SOSP '01, October 21–24, 2001, Banff, Canada.

- download performance on an Internet-wide prototype deployment as fast as standard FTP;
- provable efficiency and provably fast recovery times after failure;
- simple algorithms to achieve the above results.

CFS is not yet in operational use, and such use will likely prompt refinements to its design. One potential area for improvement is the ability of the Chord lookup algorithm to tolerate malicious participants, by verifying the routing information received from other servers. Another area that CFS does not currently address is anonymity; it is expected that anonymity, if needed, would be layered on top of the basic CFS system.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 outlines the design and design goals of CFS. Section 4 describes the Chord location protocol. Section 5 presents the detailed design of CFS. Section 6 describes implementation details, and Section 7 presents experimental results. Section 8 discusses open issues and future work. Finally, Section 9 summarizes our findings and concludes the paper.

2. Related Work

CFS was inspired by Napster [20], Gnutella [11], and particularly Freenet [6]. CFS uses peer-to-peer distributed hashing similar in spirit to a number of ongoing research projects [26, 29, 35]. In comparison to existing peer-to-peer file sharing systems, CFS offers simplicity of implementation and high performance without compromising correctness. CFS balances server load, finds data quickly for clients, and guarantees data availability in the face of server failures with very high probability. CFS, as a complete system, has individual aspects in common with many existing systems. The major relationships are summarized below.

2.1 Naming and Authentication

CFS authenticates data by naming it with public keys or content-hashes, as do many other distributed storage systems [9, 6, 7, 13, 29, 34]. The use of content-hashes to securely link together different pieces of data is due to Merkle [18]; the use of public keys to authentically name data is due to the SFS system [17].

CFS adopts naming, authentication, and file system structure ideas from SFSRO [9], which implements a secure distributed read-only file system—that is, a file system in which files can be modified only by their owner, and only through complete replacement of the file. However, SFSRO and CFS have significant differences at the architectural and mechanism levels. SFSRO defines protocols and authentication mechanisms which a client can use to retrieve data from a given server. CFS adds the ability to *dynamically* find the server currently holding the desired data, via the Chord location service. This increases the robustness and the availability of CFS, since changes in the set of servers are transparent to clients.

2.2 Peer-to-Peer Search

Napster [20] and Gnutella [11] are arguably the most widely used peer-to-peer file systems today. They present a keyword search interface to clients, rather than retrieving uniquely identified data. As a result they are more like search engines than distributed hash tables, and they trade scalability for this power: Gnutella broadcasts search queries to many machines, and Napster performs searches at a central facility. CFS as described in this paper doesn't provide search, but we are developing a scalable distributed search engine for CFS.

Mojo Nation [19] is a broadcast query peer-to-peer storage system which divides files into blocks and uses a secret sharing algorithm to distribute the blocks to a number of hosts. CFS also divides files into blocks but does not use secret sharing.

2.3 Anonymous Storage

Freenet [5] uses probabilistic routing to preserve the anonymity of clients, publishers, and servers. The anonymity requirement limits Freenet's reliability and performance. Freenet avoids associating a document with any predictable server, and avoids forming any globally coherent topology among servers. The former means that unpopular documents may simply disappear from the system, since no server has the responsibility for maintaining replicas. The latter means that a search may need to visit a large fraction of the Freenet network. As an example, Hong shows in his Figure 14-12 that in a network with 1,000 servers, the lookup path length can exceed 90 hops [23]. This means that if the hop count is limited to 90, a lookup may fail even though the document is available. Because CFS does not try to provide anonymity, it can guarantee much tighter bounds on lookup cost; for example, in a 4,096-node system, lookups essentially never exceed 10 hops.

CFS's caching scheme is similar to Freenet's in the sense that both leave cached copies of data along the query path from client to where the data was found. Because CFS finds data in significantly fewer hops than Freenet, and CFS' structured lookup paths are more likely to overlap than Freenet's, CFS can make better use of a given quantity of cache space.

Like Freenet, Publius [34] focuses on anonymity, but achieves it with encryption and secret sharing rather than routing. Publius requires a static, globally-known list of servers; it stores each share at a fixed location that is predictable from the file name. Free Haven [7] uses both cryptography and routing (using re-mailers [4]) to provide anonymity; like Gnutella, Free Haven finds data with a global search.

CFS does not attempt to provide anonymity, focusing instead on efficiency and robustness. We believe that intertwining anonymity with the basic data lookup mechanism interferes with correctness and performance. On the other hand, given a robust location and storage layer, anonymous client access to CFS could be provided by separate anonymizing proxies, using techniques similar to those proposed by Chaum [4] or Reiter and Rubin [27].

2.4 Peer-to-Peer Hash Based Systems

CFS layers storage on top of an efficient distributed hash lookup algorithm. A number of recent peer-to-peer systems use similar approaches with similar scalability and performance, including CAN [26], PAST [28, 29], OceanStore [13, 35], and Ohaha [22]. A detailed comparison of these algorithms can be found in [31].

The PAST [29] storage system differs from CFS in its approach to load balance. Because a PAST server stores whole files, a server might not have enough disk space to store a large file even though the system as a whole has sufficient free space. A PAST server solves this by offloading files it is responsible for to servers that do have spare disk space. PAST handles the load of serving popular files by caching them along the lookup path.

CFS stores blocks, rather than whole files, and spreads blocks evenly over the available servers; this prevents large files from causing unbalanced use of storage. CFS solves the related problem of different servers having different amounts of storage space with a mechanism called *virtual servers*, which gives server managers control over disk space consumption. CFS' block storage granularity helps it handle the load of serving popular large files, since the serving load is spread over many servers along with the blocks.

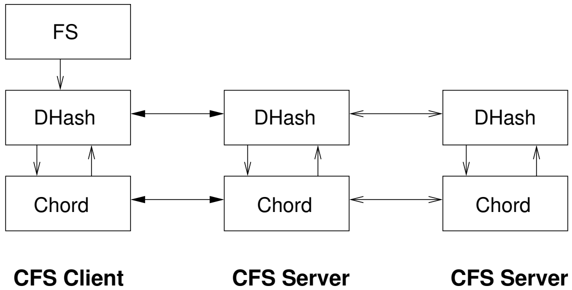

Figure 1: CFS software structure. Vertical links are local APIs; horizontal links are RPC APIs.

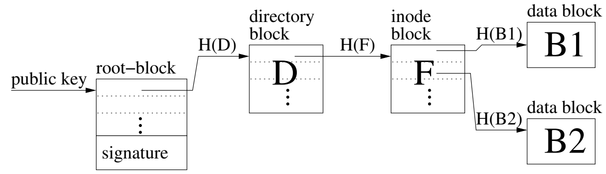

Figure 2: A simple CFS file system structure example. The root-block is identified by a public key and signed by the corresponding private key. The other blocks are identified by cryptographic hashes of their contents.

This is more space-efficient, for large files, than whole-file caching. CFS relies on caching only for files small enough that distributing blocks is not effective. Evaluating the performance impact of block storage granularity is one of the purposes of this paper.

OceanStore [13] aims to build a global persistent storage utility. It provides data privacy, allows client updates, and guarantees durable storage. However, these features come at a price: complexity. For example, OceanStore uses a Byzantine agreement protocol for conflict resolution, and a complex protocol based on Plaxton trees [24] to implement the location service [35]. OceanStore assumes that the core system will be maintained by commercial providers.

Ohaha [22] uses consistent hashing to map files and keyword queries to servers, and a Freenet-like routing algorithm to locate files. As a result, it shares some of the same weaknesses as Freenet.

2.5 Web Caches

Content distribution networks (CDNs), such as Akamai [1], handle high demand for data by distributing replicas on multiple servers. CDNs are typically managed by a central entity, while CFS is built from resources shared and owned by a cooperative group of users.

There are several proposed scalable cooperative Web caches [3, 8, 10, 15]. To locate data, these systems either multicast queries or require that some or all servers know about all other servers. As a result, none of the proposed methods is both highly scalable and robust. In addition, load balance is hard to achieve as the content of each cache depends heavily on the query pattern.

Cache Resolver [30], like CFS, uses consistent hashing to *evenly* map stored data among the servers [12, 14]. However, Cache Resolver assumes that clients know the entire set of servers; maintaining an up-to-date server list is likely to be difficult in a large peer-to-peer system where servers join and depart at unpredictable times.

3. Design Overview

CFS provides distributed read-only file storage. It is structured as a collection of servers that provide block-level storage. Publishers (producers of data) and clients (consumers of data) layer file system semantics on top of this block store much as an ordinary file system is layered on top of a disk. Many unrelated publishers may store separate file systems on a single CFS system; the CFS design is intended to support the possibility of a single world-wide system consisting of millions of servers.

3.1 System Structure

Figure 1 illustrates the structure of the CFS software. Each CFS client contains three software layers: a file system client, a DHash storage layer, and a Chord lookup layer. The client file system uses the DHash layer to retrieve blocks. The client DHash layer uses the client Chord layer to locate the servers that hold desired blocks.

Each CFS server has two software layers: a DHash storage layer and a Chord layer. The server DHash layer is responsible for storing keyed blocks, maintaining proper levels of replication as servers come and go, and caching popular blocks. The server DHash and Chord layers interact in order to integrate looking up a block identifier with checking for cached copies of the block. CFS servers are oblivious to file system semantics: they simply provide a distributed block store.

CFS clients interpret DHash blocks in a file system format adopted from SFSRO [9]; the format is similar to that of the UNIX V7 file system, but uses DHash blocks and block identifiers in place of disk blocks and disk addresses. As shown in Figure 2, each block is either a piece of a file or a piece of file system meta-data, such as a directory. The maximum size of any block is on the order of tens of kilobytes. A parent block contains the identifiers of its children.

The publisher inserts the file system's blocks into the CFS system, using a hash of each block's content (a content-hash) as its identifier. Then the publisher signs the root block with his or her private key, and inserts the root block into CFS using the corresponding public key as the root block's identifier. Clients name a file system using the public key; they can check the integrity of the root block using that key, and the integrity of blocks lower in the tree with the content-hash identifiers that refer to those blocks. This approach guarantees that clients see an authentic and internally consistent view of each file system, though under some circumstances a client may see an old version of a recently updated file system.

A CFS file system is read-only as far as clients are concerned. However, a file system may be updated by its publisher. This involves updating the file system's root block in place, to make it point to the new data. CFS authenticates updates to root blocks by checking that the new block is signed by the same key as the old block. A timestamp prevents replays of old updates. CFS allows file systems to be updated without changing the root block's identifier so that external references to data need not be changed when the data is updated.

CFS stores data for an agreed-upon finite interval. Publishers that want indefinite storage periods can periodically ask CFS for an extension; otherwise, a CFS server may discard data whose guaranteed period has expired. CFS has no explicit delete operation: instead, a publisher can simply stop asking for extensions. In this area, as in its replication and caching policies, CFS relies on the assumption that large amounts of spare disk space are available.

3.2 CFS Properties

CFS provides consistency and integrity of file systems by adopting the SFSRO file system format. CFS extends SFSRO by providing the following desirable desirable properties:

- **Decentralized control.** CFS servers need not share any administrative relationship with publishers. CFS servers could be ordinary Internet hosts whose owners volunteer spare storage and network resources.
- **Scalability.** CFS lookup operations use space and messages at most logarithmic in the number of servers.
- **Availability.** A client can always retrieve data as long as it is not trapped in a small partition of the underlying network, and as long as one of the data's replicas is reachable using the underlying network. This is true even if servers are constantly joining and leaving the CFS system. CFS places replicas on servers likely to be at unrelated network locations to ensure independent failure.
- **Load balance.** CFS ensures that the burden of storing and serving data is divided among the servers in rough proportion to their capacity. It maintains load balance even if some data are far more popular than others, through a combination of caching and spreading each file's data over many servers.
- **Persistence.** Once CFS commits to storing data, it keeps it available for at least an agreed-on interval.
- **Quotas.** CFS limits the amount of data that any particular IP address can insert into the system. This provides a degree of protection against malicious attempts to exhaust the system's storage.
- **Efficiency.** Clients can fetch CFS data with delay comparable to that of FTP, due to CFS' use of efficient lookup algorithms, caching, pre-fetching, and server selection.

The next two sections present Chord and DHash, which together provide these properties.

4. Chord Layer

CFS uses the Chord protocol to locate blocks [31]. Chord supports just one operation: given a key, it will determine the node responsible for that key. Chord does not itself store keys and values, but provides primitives that allow higher-layer software to build a wide variety of storage systems; CFS is one such use of the Chord primitive. The rest of this section summarizes Chord and describes new algorithms for robustness and server selection to support applications such as CFS.

4.1 Consistent Hashing

Each Chord node has a unique m-bit node identifier (ID), obtained by hashing the node's IP address and a *virtual node index*. Chord views the IDs as occupying a circular identifier space. Keys are also mapped into this ID space, by hashing them to m-bit key IDs. Chord defines the node responsible for a key to be the *successor* of that key's ID. The *successor* of an ID j is the node with the smallest ID that is greater than or equal to j (with wrap-around), much as in consistent hashing [12].

Consistent hashing lets nodes enter and leave the network with minimal movement of keys. To maintain correct successor mappings when a node n joins the network, certain keys previously assigned to n's successor become assigned to n. When node n leaves the network, all of n's assigned keys are reassigned to its successor. No other changes in the assignment of keys to nodes need occur.

Consistent hashing is straightforward to implement, with constant-time lookups, if all nodes have an up-to-date list of all other nodes. However, such a system does not scale; Chord provides a scalable, distributed version of consistent hashing.

4.2 The Chord Lookup Algorithm

A Chord node uses two data structures to perform lookups: a successor list and a finger table. Only the successor list is required for correctness, so Chord is careful to maintain its accuracy. The finger table accelerates lookups, but does not need to be accurate, so Chord is less aggressive about maintaining it. The following discussion first describes how to perform correct (but slow) lookups with the successor list, and then describes how to accelerate them up with the finger table. This discussion assumes that there are no malicious participants in the Chord protocol; while we believe that it should be possible for nodes to verify the routing information that other Chord participants send them, the algorithms to do so are left for future work.

Every Chord node maintains a list of the identities and IP addresses of its r immediate successors on the Chord ring. The fact that every node knows its own successor means that a node can always process a lookup correctly: if the desired key is between the node and its successor, the latter node is the key's successor; otherwise the lookup can be forwarded to the successor, which moves the lookup strictly closer to its destination.

A new node n learns of its successors when it first joins the Chord ring, by asking an existing node to perform a lookup for n's successor; n then asks that successor for its successor list. The r entries in the list provide fault tolerance: if a node's immediate successor does not respond, the node can substitute the second entry in its successor list. All r successors would have to simultaneously fail in order to disrupt the Chord ring, an event that can be made very improbable with modest values of r. An implementation should use a fixed r, chosen to be $2\log_2 N$ for the foreseeable maximum number of nodes N.

The main complexity involved with successor lists is in notifying an existing node when a new node should be its successor. The stabilization procedure described in [31] does this in a way that guarantees to preserve the connectivity of the Chord ring's successor pointers.

Lookups performed only with successor lists would require an average of $N/2$ message exchanges, where N is the number of servers. To reduce the number of messages required to $O(\log N)$, each node maintains a finger table table with m entries. The i^{th} entry in the table at node n contains the identity of the *first* node that succeeds n by at least 2^{i-1} on the ID circle. Thus every node knows the identities of nodes at power-of-two intervals on the ID circle from its own position. A new node initializes its finger table by querying an existing node. Existing nodes whose finger table or successor list entries should refer to the new node find out about it by periodic lookups.

Figure 3 shows pseudo-code to look up the successor of identifier id. The main loop is in *find_predecessor*, which sends *preceding_node_list* RPCs to a succession of other nodes; each RPC searches the tables of the other node for nodes yet closer to id. Because finger table entries point to nodes at power-of-two intervals around the ID ring, each iteration will set n' to a node halfway on the ID ring between the current n' and id. Since *preceding_node_list* never returns an ID greater than id, this process will never overshoot the correct successor. It may under-shoot, especially if a new node has recently joined with an ID just before id; in that case the check for $id \notin (n', n'.\mathit{successor}]$ ensures that

```
// Ask node n to find id's successor; first
// finds id's predecessor, then asks that
// predecessor for its own successor.
n.find_successor(id)
  n' = find_predecessor(id);
  return n'.successor();

// Ask node n to find id's predecessor.
n.find_predecessor(id)
  n' = n;
  while (id ∉ (n', n'.successor()])
    l = n'.preceding_node_list(id);
    n' = max n'' ∈ l s.t. n'' is alive
  return n';

// Ask node n for a list of nodes in its finger table or
// successor list that precede id.
n.preceding_node_list(id)
  return {n' ∈ {fingers ∪ successors}
    s.t. n' ∈ (n, id]}
```

Figure 3: The pseudo-code to find the successor node of an identifier id. Remote procedure calls are preceded by the remote node.

find_predecessor persists until it finds a pair of nodes that straddle id.

Two aspects of the lookup algorithm make it robust. First, an RPC to *preceding_node_list* on node n returns a list of nodes that n believes are between it and the desired id. Any one of them can be used to make progress towards the successor of id; they must all be unresponsive for a lookup to fail. Second, the **while** loop ensures that *find_predecessor* will keep trying as long as it can find any next node closer to id. As long as nodes are careful to maintain correct successor pointers, *find_predecessor* will eventually succeed.

In the usual case in which most nodes have correct finger table information, each iteration of the **while** loop eliminates half the remaining distance to the target. This means that the hops early in a lookup travel long distances in the ID space, and later hops travel small distances. The efficacy of the caching mechanism described in Section 5.2 depends on this observation.

The following two theorems, proved in an accompanying technical report [32], show that neither the success nor the performance of Chord lookups is likely to be affected even by massive simultaneous failures. Both theorems assume that the successor list has length $r = O(\log N)$. A Chord ring is *stable* if every node's successor list is correct.

THEOREM 1. *In a network that is initially stable, if every node then fails with probability 1/2, then with high probability find_successor returns the closest living successor to the query key.*

THEOREM 2. *In a network that is initially stable, if every node then fails with probability 1/2, then the expected time to execute find_successor is* $O(\log N)$.

The evaluation in Section 7 validates these theorems experimentally.

4.3 Server Selection

Chord reduces lookup latency by preferentially contacting nodes likely to be nearby in the underlying network. This server selection was added to Chord as part of work on CFS, and was not part of Chord as originally published.

At each step in *find_predecessor(id)* (Figure 3), the node doing the lookup (n) can choose the next hop from a set of nodes. Initially this set is the contents of n's own routing tables; subsequently the set is the list of nodes returned by the *preceding_node_list* RPC to the previous hop (m). Node m tells n the measured latency to each node in the set; m collected these latencies when it acquired its finger table entries. Different choices of next-hop node will take the query different distances around the ID ring, but impose different RPC latencies; the following calculation seeks to pick the best combination.

Chord estimates the total cost $C(n_i)$ of using each node in the set of potential next hops:

$$\begin{aligned} C(n_i) &= d_i + \overline{d} \times H(n_i) \\ H(n_i) &= \text{ones}((n_i - id) >> (160 - \log N)) \end{aligned}$$

$H(n_i)$ is an estimate of the number of Chord hops that would remain after contacting n_i. N is node n's estimate of the total number of Chord nodes in the system, based on the density of nodes nearby on the ID ring. $\log N$ is an estimate of the number of *significant* high bits in an ID; id and its successor are likely to agree in these bits, but not in less significant bits. $(n_i - id) >> (160 - \log N)$ yields just the significant bits in the ID-space distance between n_i and the target key id, and the *ones* function counts how many bits are set in that difference; this is approximately the number of Chord hops between n_i and id. Multiplying by $\overline{d}$, the average latency of all the RPCs that node n has ever issued, estimates the time it would take to send RPCs for those hops. Adding d_i, the latency to node n_i as reported by node m, produces a complete cost estimate. Chord uses the node with minimum $C(n_i)$ as the next hop.

One benefit of this server selection method is that no extra measurements are necessary to decide which node is closest; the decisions are made based on latencies observed while building finger tables. However, nodes rely on latency measurements taken by other nodes. This works well when low latencies from nodes a to b, and b to c, mean that latency is also low from a to c. Measurements of our Internet test-bed suggest that this is often true [33].

4.4 Node ID Authentication

If Chord nodes could use arbitrary IDs, an attacker could destroy chosen data by choosing a node ID just after the data's ID. With control of the successor, the attacker's node could effectively delete the block by denying that the block existed.

To limit the opportunity for this attack, a Chord node ID must be of the form $h(x)$, where h is the SHA-1 hash function and x is the node's IP address concatenated with a virtual node index. The virtual node index must fall between 0 and some small maximum. As a result, a node cannot easily control the choice of its own Chord ID.

When a new node n joins the system, some existing nodes may decide to add n to their finger tables. As part of this process, each such existing node sends a message to n's claimed IP address containing a nonce. If the node at that IP address admits to having n's ID, and the claimed IP address and virtual node index hash to the ID, then the existing node accepts n.

With this defense in place, an attacker would have to control roughly as many IP addresses as there are total other nodes in the Chord system in order to have a good chance of targeting arbitrary blocks. However, owners of large blocks of IP address space tend to be more easily identifiable (and less likely to be malicious) than individuals.

5. DHash Layer

The CFS DHash layer stores and retrieves uniquely identified blocks, and handles distribution, replication, and caching of those blocks. DHash uses Chord to help it locate blocks.

DHash reflects a key CFS design decision: to split each file system (and file) into blocks and distribute those blocks over many servers. This arrangement balances the load of serving popular files over many servers. It also increases the number of messages required to fetch a whole file, since a client must look up each block separately. However, the network bandwidth consumed by a lookup is small compared to the bandwidth required to deliver the block. In addition, CFS hides the block lookup latency by pre-fetching blocks.

Systems such as Freenet [6] and PAST [29] store whole files. This results in lower lookup costs than CFS, one lookup per file rather than per block, but requires more work to achieve load balance. Servers unlucky enough to be responsible for storing very large files may run out of disk space even though the system as a whole has sufficient free space. Balancing the load of serving whole files typically involves adaptive caching. Again, this may be awkward for large files; a popular file must be stored in its entirety at each caching server. DHash also uses caching, but only depends on it for small files.

DHash's block granularity is particularly well suited to serving large, popular files, such as software distributions. For example, in a 1,000-server system, a file as small as 8 megabytes will produce a reasonably balanced serving load with 8 KByte blocks. A system that balances load by caching whole files would require, in this case, about 1,000 times as much total storage to achieve the same load balance. On the other hand, DHash is not as efficient as a whole-file scheme for large but unpopular files, though the experiments in Section 7.1 show that it can provide competitive download speeds. DHash's block granularity is not likely to affect (for better or worse) performance or load balance for small files. For such files, DHash depends on caching and on server selection among block replicas (described in Section 5.1).

Table 1 shows the API that the DHash layer exposes. The CFS file system client layer uses `get` to implement application requests to open files, read files, navigate directories, etc. Publishers of data use a special application that inserts or updates a CFS file system using the `put_h` and `put_s` calls.

5.1 Replication

DHash replicates each block on k CFS servers to increase availability, maintains the k replicas automatically as servers come and go, and places the replicas in a way that clients can easily find them.

DHash places a block's replicas at the k servers immediately after the block's successor on the Chord ring (see Figure 4). DHash can easily find the identities of these servers from Chord's r-entry successor list. CFS must be configured so that $r \geq k$. This placement of replicas means that, after a block's successor server fails, the block is immediately available at the block's new successor.

The DHash software in a block's successor server manages replication of that block, by making sure that all k of its successor servers have a copy of the block at all times. If the successor server fails, the block's new successor assumes responsibility for the block.

The value of this replication scheme depends in part on the independence of failure and unreachability among a block's k replica servers. Servers close to each other on the ID ring are not likely to be physically close to each other, since a server's ID is based on a hash of its IP address. This provides the desired independence of failure.

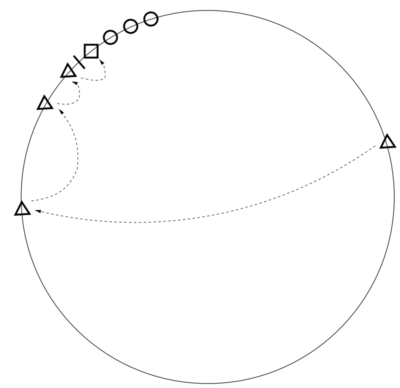

Figure 4: The placement of an example block's replicas and cached copies around the Chord identifier ring. The block's ID is shown with a tick mark. The block is stored at the successor of its ID, the server denoted with the square. The block is replicated at the successor's immediate successors (the circles). The hops of a typical lookup path for the block are shown with arrows; the block will be cached at the servers along the lookup path (the triangles).

CFS could save space by storing coded pieces of blocks rather than whole-block replicas, using an algorithm such as IDA [25]. CFS doesn't use coding, because storage space is not expected to be a highly constrained resource.

The placement of block replicas makes it easy for a client to select the replica likely to be fastest to download. The result of the Chord lookup for block id is the identity of the server that immediately precedes id. The client asks this predecessor for its successor list, which will include the identities of the servers holding replicas of block id as well as latency measurements between the predecessor and these servers. The client then fetches the block from the replica with the lowest reported latency. As with the Chord server selection described in Section 4.3, this approach works best when proximity in the underlying network is transitive. Fetching from the lowest-latency replica has the side-effect of spreading the load of serving a block over the replicas.

5.2 Caching

DHash caches blocks to avoid overloading servers that hold popular data. Each DHash layer sets aside a fixed amount of disk storage for its cache. When a CFS client looks up a block key, it performs a Chord lookup, visiting intermediate CFS servers with IDs successively closer to that of the key's successor (see Figure 4). At each step, the client asks the intermediate server whether it has the desired block cached. Eventually the client arrives at either the key's successor or at an intermediate server with a cached copy. The client then sends a copy of the block to each of the servers it contacted along the lookup path. Future versions of DHash will send a copy to just the second-to-last server in the path traversed by the lookup; this will reduce the amount of network traffic required per lookup, without significantly decreasing the caching effectiveness.

Since a Chord lookup takes shorter and shorter hops in ID space as it gets closer to the target, lookups from different clients for the same block will tend to visit the same servers late in the lookup. As a result, the policy of caching blocks along the lookup path is likely to be effective.

DHash replaces cached blocks in least-recently-used order. Copies of a block at servers with IDs far from the block's succes-

Function	Description
`put_h(block)`	Computes the block's key by hashing its contents, and sends it to the key's successor server for storage.
`put_s(block, pubkey)`	Stores or updates a signed block; used for root blocks. The block must be signed with the given public key. The block's Chord key will be the hash of `pubkey`.
`get(key)`	Fetches and returns the block associated with the specified Chord key.

Table 1: DHash client API; exposed to client file system software.

sor are likely to be discarded first, since clients are least likely to stumble upon them. This has the effect of preserving the cached copies close to the successor, and expands and contracts the degree of caching for each block according to its popularity.

While caching and replication are conceptually similar, DHash provides them as distinct mechanisms. DHash stores replicas in predictable places, so it can ensure that enough replicas always exist. In contrast, the number of cached copies cannot easily be counted, and might fall to zero. If fault-tolerance were achieved solely through cached copies, an unpopular block might simply disappear along with its last cached copy.

CFS avoids most cache consistency problems because blocks are keyed by content hashes. Root blocks, however, use public keys as identifiers; a publisher can change a root block by inserting a new one signed with the corresponding private key. This means that cached root blocks may become stale, causing some clients to see an old, but internally consistent, file system. A client can check the freshness of a cached root block [9] to decide whether to look for a newer version. Non-root blocks that no longer have any references to them will eventually be eliminated from caches by LRU replacement.

5.3 Load Balance

DHash spreads blocks evenly around the ID space, since the content hash function uniformly distributes block IDs. If each CFS server had one ID, the fact that IDs are uniformly distributed would mean that every server would carry roughly the same storage burden. This is not desirable, since different servers may have different storage and network capacities. In addition, even uniform distribution doesn't produce perfect load balance; the maximum storage burden is likely to be about $\log(N)$ times the average due to irregular spacing between server IDs [12].

To accommodate heterogeneous server capacities, CFS uses the notion (from [12]) of a real server acting as multiple *virtual servers*. The CFS protocol operates at the virtual server level. A virtual server uses a Chord ID that is derived from hashing both the real server's IP address and the index of the virtual server within the real server.

A CFS server administrator configures the server with a number of virtual servers in rough proportion to the server's storage and network capacity. This number can be adjusted from time to time to reflect observed load levels.

Use of virtual servers could potentially increase the number of hops in a Chord lookup. CFS avoids this expense by allowing virtual servers on the same physical server to examine each others' tables; the fact that these virtual servers can take short-cuts through each others' routing tables exactly compensates for the increased number of servers.

CFS could potentially vary the number of virtual servers per real server adaptively, based on current load. Under high load, a real server could delete some of its virtual servers; under low load, a server could create additional virtual servers. Any such algorithm would need to be designed for stability under high load. If a server is overloaded because the CFS system as a whole is overloaded, then automatically deleting virtual servers might cause a cascade of such deletions.

5.4 Quotas

The most damaging technical form of abuse that CFS is likely to encounter is malicious injection of large quantities of data. The aim of such an attack might be to use up all the disk space on the CFS servers, leaving none available for legitimate data. Even a non-malicious user could cause the same kind of problem by accident.

Ideally, CFS would impose per-publisher quotas based on reliable identification of publishers, as is done in the PAST system [29]. Reliable identification usually requires some form of centralized administration, such as a certificate authority. As a decentralized approximation, CFS bases quotas on the IP address of the publisher. For example, if each CFS server limits any one IP address to using 0.1% of its storage, then an attacker would have to mount an attack from about 1,000 machines for it to be successful. This mechanism also limits the storage used by each legitimate publisher to just 0.1%, assuming each publisher uses just one IP address.

This limit is not easy to subvert by simple forging of IP addresses, since CFS servers require that publishers respond to a confirmation request that includes a random nonce, as described in Section 4.4. This approach is weaker than one that requires publishers to have unforgeable identities, but requires no centralized administrative mechanisms.

If each CFS server imposes a fixed per-IP-address quota, then the total amount of storage an IP address can consume will grow linearly with the total number of CFS servers. It may prove desirable to enforce a fixed quota on total storage, which would require the quota imposed by each server to decrease in proportion to the total number of servers. An adaptive limit of this form is possible, using the estimate of the total number of servers that the Chord software maintains.

5.5 Updates and Deletion

CFS allows updates, but in a way that allows only the publisher of a file system to modify it. A CFS server will accept a request to store a block under either of two conditions. If the block is marked as a content-hash block, the server will accept the block if the supplied key is equal to the SHA-1 hash of the block's content. If the block is marked as a signed block, the block must be signed by a public key whose SHA-1 hash is the block's CFS key.

The low probability of finding two blocks with the same SHA-1 hash prevents an attacker from changing the block associated with a content-hash key, so no explicit protection is required for most of a file system's blocks. The only sensitive block is a file system's root block, which is signed; its safety depends on the publisher avoiding disclosure of the private key.

CFS does not support an explicit delete operation. Publishers must periodically refresh their blocks if they wish CFS to continue to store them. A CFS server may delete blocks that have not been refreshed recently.

One benefit of CFS' implicit deletion is that it automatically recovers from malicious insertions of large quantities of data. Once the attacker stops inserting or refreshing the data, CFS will gradually delete it.

6. Implementation

CFS is implemented in 7,000 lines of C++, including the 3,000 line Chord implementation. It consists of a number of separate programs that run at user level. The programs communicate over UDP with a C++ RPC package provided by the SFS toolkit [16]. A busy CFS server may exchange short messages with a large number of other servers, making the overhead of TCP connection setup unattractive compared to UDP. The internal structure of each program is based on asynchronous events and callbacks, rather than threads. Each software layer is implemented as a library with a C++ interface. CFS runs on Linux, OpenBSD, and FreeBSD.

6.1 Chord Implementation

The Chord library maintains the routing tables described in Section 4. It exports these tables to the DHash layer, which implements its own integrated version of the Chord lookup algorithm. The implementation uses the SHA-1 cryptographic hash function to produce CFS block identifiers from block contents. This means that block and server identifiers are 160 bits wide.

The Chord implementation maintains a running estimate of the total number of Chord servers, for use in the server selection algorithm described in Section 4.3. Each server computes the fraction of the ID ring that the r nodes in its successor list cover; let that fraction be f. Then the estimated total number of servers in the system is r/f.

6.2 DHash Implementation

DHash is implemented as a library which depends on Chord. Each DHash instance is associated with a Chord virtual server and communicates with that virtual server through a function call interface. DHash instances on different servers communicate with one another via RPC.

DHash has its own implementation of the Chord lookup algorithm, but relies on the Chord layer to maintain the routing tables. Integrating block lookup into DHash increases its efficiency. If DHash instead called the Chord *find_successor* routine, it would be awkward for DHash to check each server along the lookup path for cached copies of the desired block. It would also cost an unneeded round trip time, since both Chord and DHash would end up separately contacting the block's successor server.

Pseudo-code for the DHash implementation of *lookup(key)* is shown in Figure 5; this is DHash's version of the Chord code shown in Figure 3. The function *lookup(key)* returns the data associated with *key* or an error if it cannot be found. The function *lookup* operates by repeatedly invoking the remote procedure *n'.lookup_step(key)* which returns one of three possible values. If the called server (n') stores or caches the data associated with *key*, then *n'.lookup_step(key)* returns that data. If n' does not store the data, then *n'.lookup_step(key)* returns instead the closest predecessor of *key* (determined by consulting local routing tables on n'). Finally, *n'.lookup_step* returns an error if n' is the true successor of the key but does not store its associated data.

If *lookup* tries to contact a failed server, the RPC machinery will return `RPC_FAILURE`. The function *lookup* then backtracks to the previously contacted server, tells it about the failed server with *alert*, and asks it for the next-best predecessor. At some point *lookup(key)* will have contacted a pair of servers on either side of *key*. If there have been server failures, the second server of the pair may not be the *key*'s original successor. However, that second server will be the first live successor, and will hold a replica for *key*, assuming that not all of its replicas have failed.

Though the pseudo-code does not show it, the virtual servers on any given physical server look at each others' routing tables and block stores. This allows lookups to progress faster around the ring, and increases the chances of encountering a cached block.

```
// RPC handler on server n. Returns block with ID key,
// or the best next server to talk to.
n.lookup_step(key)
  if key ∈ (stored ∪ cached ∪ replicated) then
    return[COMPLETE, key, data_key]
  else if key ∈ (predecessor, myid]
    return NONEXISTENT
  else if key ∈ (myid, first live successor]
    next_hop = first live successor
  else
    // Find highest server < key in my finger table or successor list.
    next_hop = lookup_closest_pred(key)
  succ_list = {s ∈ {fingers ∪ successors} s.t. s > next_hop}
  return [CONTINUE, next_hop, succ_list]

// RPC handler to ask the Chord software to delete
// server id from the finger list and successor list.
n.alert(id)

// Return the block associated with key, or an error.
// Runs on the server that invokes lookup().
lookup(key)
  p.push(n) // A stack to accumulate the path.
  [status, res] = n.lookup_step(key)
  repeat
    if (status = COMPLETE)
      return res.data_key
    else if (status = CONTINUE)
      if (res.next_hop = p.top)
        // p.top knew no server other than itself.
        return NONEXISTENT
      else if (key ∈ (p.top', p.top])
        // p.top should have had the block.
        return NONEXISTENT
      else // explore next hop
        p.push(res.next_hop)
        [status, res] = res.next_hop.lookup_step(key)
    else if (status = RPC_FAILURE)
      // Try again at previous hop.
      failed = p.pop()
      last = p.top()
      last.alert(failed)
      [status, res] = last.lookup_step(key)
    else
      return NONEXISTENT
```

Figure 5: The procedure used by DHash to locate a block.

6.3 Client Implementation

The CFS client software layers a file system on top of DHash. CFS exports an ordinary UNIX file system interface by acting as a local NFS server using the SFS user level file system toolkit [16]. A CFS client runs on the same machine as CFS server; the client communicates with the local server via a UNIX domain socket and uses it as a proxy to send queries to non-local CFS servers.

The DHash back end is sufficiently flexible to support a number of different client interfaces. For instance, we are currently implementing a client which acts as a web proxy in order to layer its name space on top of the name space of the world wide web.

7. Experimental Results

In order to demonstrate the practicality of the CFS design, we present two sets of tests. The first explores CFS performance on a modest number of servers distributed over the Internet, and focuses on real-world client-perceived performance. The second involves larger numbers of servers running on a single machine and focuses on scalability and robustness.

Quotas (Section 5.4) were not implemented in the tested software. Cryptographic verification of updates (Section 5.5) and server ID authentication (Section 4.4) were implemented but not enabled. This has no effect on the results presented here.

Unless noted, all tests were run with caching turned off, with no replication, with just one virtual server per physical server, and with server selection turned off. These defaults allow the effects of these features to be individually illustrated. The experiments involve only block-level DHash operations, with no file-system meta-data; the client software driving the experiments fetches a file by fetching a specified list of block identifiers. Every server maintains a successor list with $2\log_2(N)$ entries, as mentioned in Section 4, to help maintain ring connectivity. While CFS does not automatically adjust the successor list length to match the number of servers, its robustness is not sensitive to the exact value.

7.1 Real Life

The tests described in this section used CFS servers running on a testbed of 12 machines scattered over the Internet. The machines are part of the RON testbed [2]; 9 of them are at sites spread over the United States, and the other three are in the Netherlands, Sweden, and South Korea. The servers held a one megabyte CFS file split into 8K blocks. To test download speed, client software on each machine fetched the entire file. The machines fetched the file one at a time.

Three RPCs, on average, were required to fetch each block in these experiments. The client software uses pre-fetch to overlap the lookup and fetching of blocks. The client initially issues a window of some number of parallel block fetches; as each fetch completes, the client starts a new one.

Figure 6 shows the average download speeds, for a range of pre-fetch window sizes, with and without server selection. A block fetch without server selection averages about 430 milliseconds; this explains why the download speed is about 20 KBytes/second when fetching one 8KByte block at a time. Increasing the amount of pre-fetch increases the speed; for example, fetching three blocks at a time yields an average speed of about 50 KBytes/second. Large amounts of pre-fetch are counter-productive, since they can congest the client server's network connection.

Server selection increases download speeds substantially for small amounts of pre-fetch, almost doubling the download speed when no pre-fetch is used. The improvement is less dramatic for larger amounts of pre-fetch, partially because concentrating block fetches on just the nearest servers may overload those servers' links. The data shown here were obtained using a flawed version of the server selection algorithm; the correct algorithm would likely yield better download speeds.

Figure 7 shows the distribution of speeds seen by the downloads from the different machines, for different pre-fetch windows, with and without server selection. The distributions of speeds without server selection are fairly narrow: every download is likely to require a few blocks from every server, so all downloads see a similar mix of per-block times. The best download speeds were from a machine at New York University with good connections to multiple

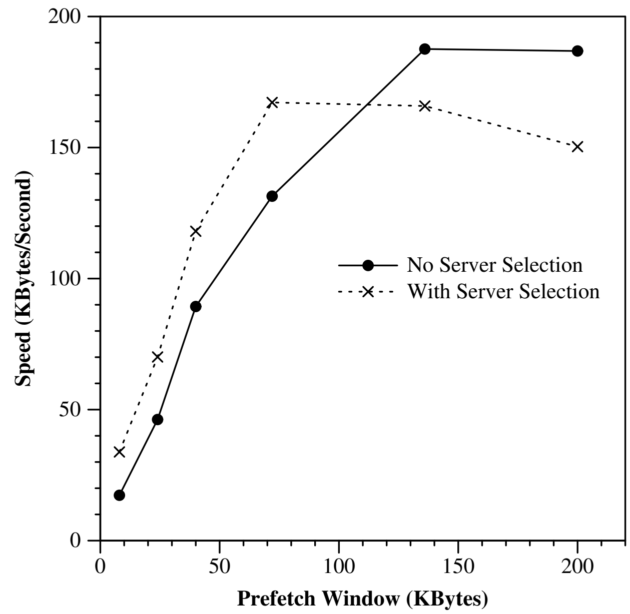

Figure 6: Download speeds achieved while fetching a one-megabyte file with CFS on the Internet testbed, for a range of pre-fetch window sizes. Each point is the average of the times seen by each testbed machine. One curve includes server selection; the other does not.

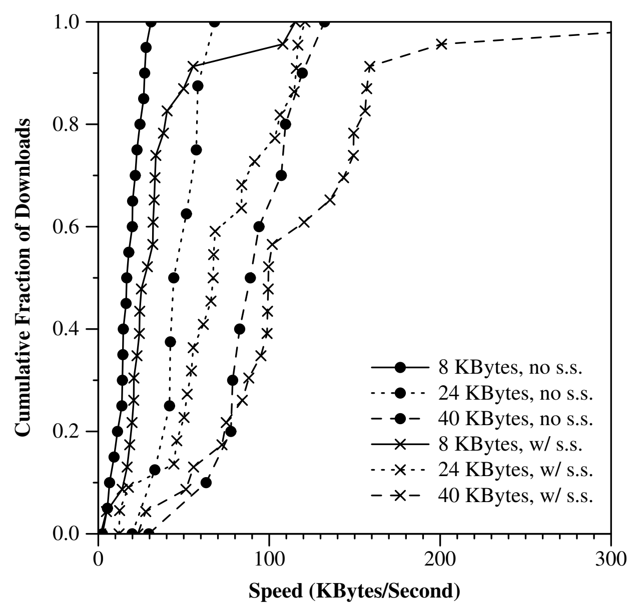

Figure 7: Cumulative distribution of the download speeds plotted in Figure 6, for various pre-fetch windows. Results with and without server selection are marked "w/ s.s." and "no s.s." respectively.

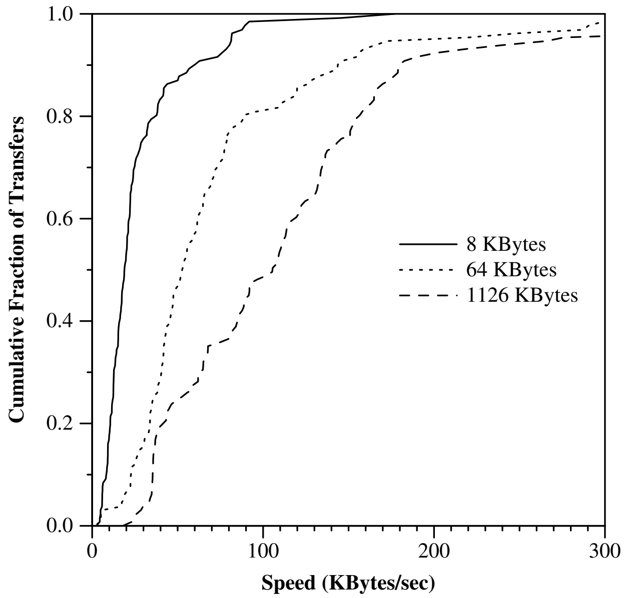

Figure 8: Distribution of download speeds achieved by ordinary TCP between each pair hosts on the Internet testbed, for three file sizes.

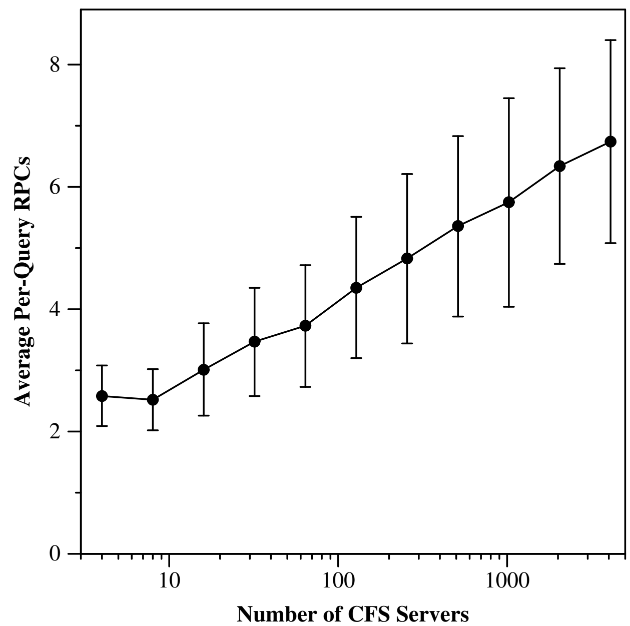

Figure 9: The average number of RPCs that a client must issue to find a block, as function of the total number of servers. The error bars reflect one standard deviation. This experimental data is linear on a log plot, and thus fits the expectation of a logarithmic growth.

backbones. The worst download speeds for small pre-fetch windows were from sites outside the United States, which have high latency to most of the servers. The worst speeds for large amounts of pre-fetch were for fetches from cable modem sites in the United States, which have limited link capacity.

The speed distributions with server selection are more skewed than without it. Most of the time, server selection improves download speeds by a modest amount. Sometimes it improves them substantially, usually for downloads initiated by well-connected sites. Sometimes server selection makes download speeds worse, usually for downloads initiated by sites outside the United States.

To show that the CFS download speeds are competitive with other file access protocols, files of various sizes were transferred between every pair of the testbed machines using ordinary TCP. The files were transferred one at a time, one whole file per TCP connection. Figure 8 shows the cumulative distribution of transfer speeds over the various machine pairs, for 8 KByte, 64 KByte, and 1.1 MByte files. The wide distributions reflect the wide range of propagation delays and link capacities between different pairs of machines. The best speeds are between well-connected sites on the east coast of the United States. The worst speeds for 8 KByte transfers occur when both end-points are outside the United States; the worst for one-megabyte transfers occur when one endpoint is outside the United States and the other is a cable modem, combining high latency with limited link speed.

CFS with a 40 KByte pre-fetch window achieves speeds competitive with TCP, on average. The CFS speeds generally have a distribution much narrower than those of TCP. This means that users are more likely to see repeatably good performance when fetching files with CFS than when fetching files from, for example, FTP servers.

7.2 Controlled Experiments

The remainder of the results were obtained from a set of CFS servers running on a single machine and using the local loopback network interface to communicate with each other. These servers act just as if they were on different machines. This arrangement is appropriate for controlled evaluation of CFS' scalability and tolerance to failure.

7.2.1 Lookup Cost

Looking up a block of data is expected to require $O(\log(N))$ RPCs. The following experiment verifies this expectation. For each of a range of numbers of servers, 10,000 blocks were inserted into the system. Then 10,000 lookups were done, from a single server, for randomly selected blocks. The number of RPCs required for each lookup was recorded. The averages are plotted in Figure 9, along with error bars showing one standard deviation.

The results are linear on a log plot, and thus fit the expectation of logarithmic growth. The actual values are about $\frac{1}{2}\log_2(N)$; for example, with 4096 servers, lookups averaged 6.7 RPCs. The number of RPCs required is determined by the number of bits in which the originating server's ID and the desired block's ID differ [31]; this will average about half of the bits, which accounts for the $\frac{1}{2}$.

7.2.2 Load Balance

One of the main goals of CFS is to balance the load over the servers. CFS achieves load balanced storage by breaking file systems up into many blocks and distributing the blocks over the servers. It further balances storage by placing multiple virtual servers per physical server, each virtual server with its own ID. We expect that $O(\log(N))$ virtual servers per physical server will be sufficient to balance the load reasonably well [31].

Figure 10 shows typical distributions of ID space among 64 physical servers for 1, 6, and 24 virtual servers per physical server. The crosses represent an actual distribution of 10,000 blocks over 64 physical servers each with 6 virtual servers. The desired result is that each server's fraction be 0.016. With only one virtual server per server (i.e., without using virtual servers), some servers would store no blocks, and others would store many times the average.

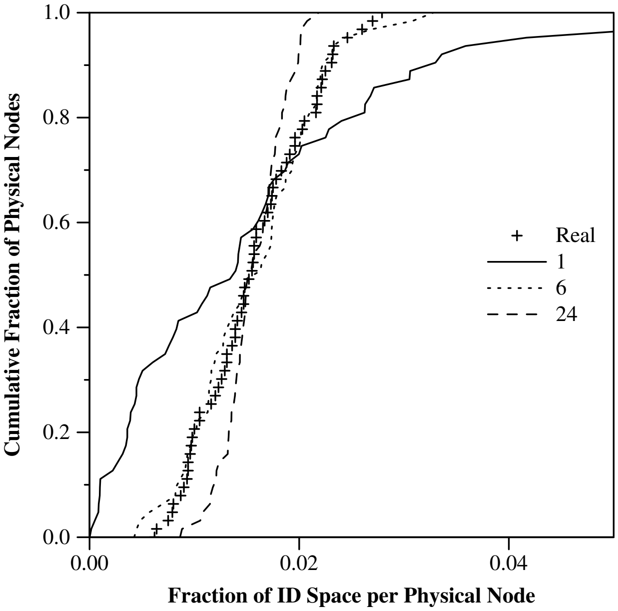

Figure 10: Representative cumulative distributions of the fraction of the key space a server might be responsible for. 64 servers are simulated, each with 1, 6, or 24 virtual servers. The data marked Real is derived from the distribution of 10,000 blocks among 64 servers, each with 6 virtual servers.

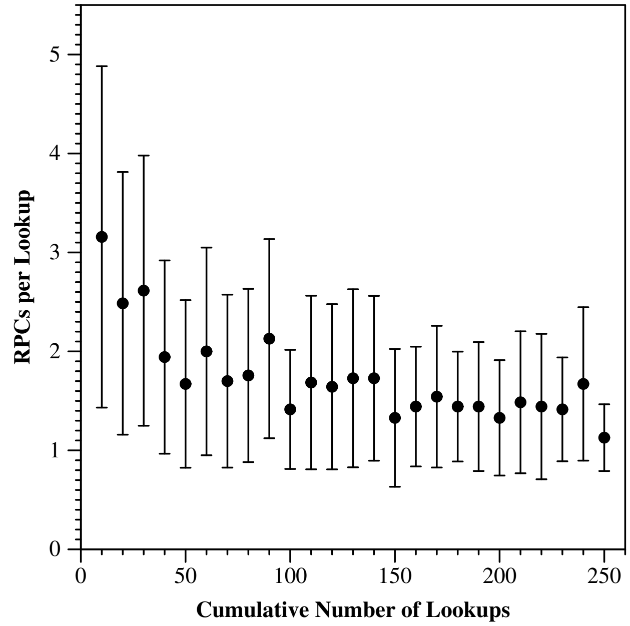

Figure 11: Impact of caching on successive client fetches of the same block. Each point is the average number of RPCs for 10 successive fetches from randomly chosen servers; the error bars indicate one standard deviation. The system has 1,000 servers.

With multiple virtual servers per server, the sum of the parts of the ID space that a server's virtual servers are responsible for is more tightly clustered around the average.

The fact that CFS spreads the storage of blocks across servers means that in many cases the burden of serving the blocks will also be evenly spread. For large files this will be true even if some files are more popular than others, since a file's blocks are widely spread. If the popular data consists of only a few blocks, then the servers that happen to be those blocks' successors will experience high load. The next section describes how caching helps balance the serving load for small files.

7.2.3 Caching

CFS caches blocks along the lookup path. As the initiating server contacts successive servers, each checks whether it already has the desired block cached. Once the initiating server has found the block, it sends a copy to each of the servers it contacted during the lookup; these servers add the block to their caches. This scheme is expected to produce high cache hit rates because the lookup paths for the same block from different sources will tend to intersect as they get closer to block's successor server.

Figure 11 illustrates how well caching works. A single block is inserted into a 1,000 server system. Then a sequence of randomly chosen servers fetch the block. The graph shows how the number of RPCs required to fetch the block decreases with the number of cumulative fetches, due to the block being cached in more places. Each plotted point is the average of 10 sequential fetches. A quirk in the implementation prevents the originating server from checking its own cache, which is why no fetches have an RPC count of zero.

As expected, the RPC counts decrease, since more and more servers have the block cached. The RPC counts decrease significantly after just a few lookups. Figure 9 shows that lookups without caching in a 1,000-server system require an average of 5.7 RPCs, while after 10 lookups with caching an average of only 3.2 hops are required. The net effect is to improve client-perceived performance and to spread the load of serving small files.

7.2.4 Storage Space Control

By varying the number of virtual servers on a physical server, a server's owner can control the amount of data that CFS stores on the server, and thus the amount that the server must serve. Figure 12 shows how effective this is. The experiment involves seven physical servers, with 1, 2, 4, 8, 16, 32, 64, and 128 virtual servers, respectively. 10,000 blocks are inserted into the system, and the relationship between how many virtual servers a physical server has and how many blocks it must store is plotted. For example, the physical server with 16 virtual server stores 586 blocks; there are a total of 255 virtual servers, so this is close to the expected value $627 = 10000 \times \frac{16}{255}$. Since the relationship of blocks to virtual servers is linear, an administrator can easily adjust a CFS server's storage consumption.

There is little memory overhead to running many virtual servers to achieve fine-grained control over load. Each virtual server requires its own finger table and successor list, as well as accounting structures for the block store and cache; the total memory footprint of these structures in our unoptimized implementation is less than 10KBytes.

7.2.5 Effect of Failure

After a CFS server fails, some time will pass before the remaining servers react to the failure, by correcting their finger tables and successor pointers and by copying blocks to maintain the desired level of replication. Theorems 1 and 2 suggest that CFS will be able to perform lookups correctly and efficiently before this recovery process starts, even in the face of massive failure.

To test this, 1,000 blocks are inserted into a 1,000-server system.

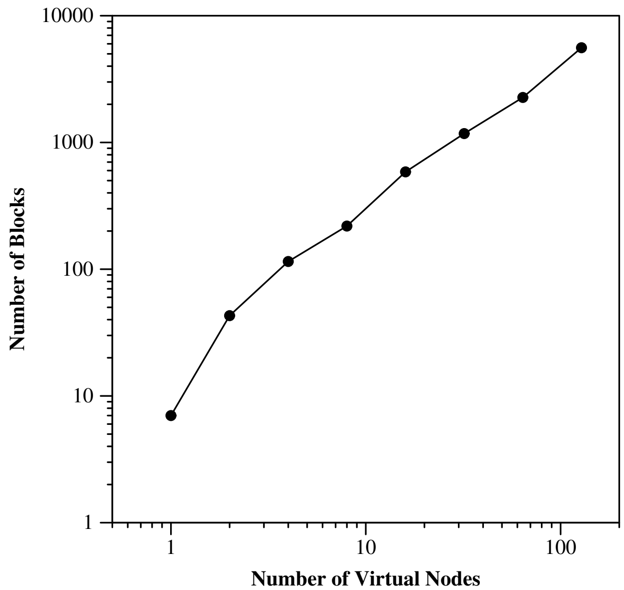

Figure 12: Impact of the number of virtual servers per physical server on the total amount of data that the physical server must store.

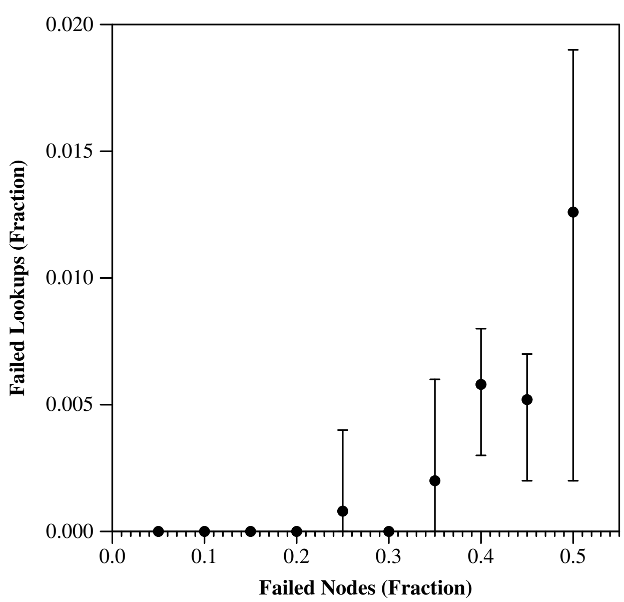

Figure 13: Fraction of block request failures as a function of the fraction of 1,000 CFS servers that fail. Each data point is the average of 5 experiments involving 1,000 block lookups; the error bars indicate the minimum and maximum results.

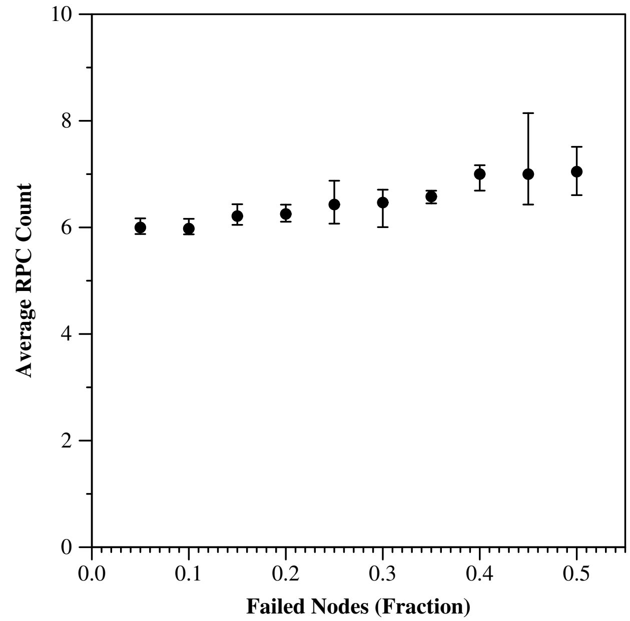

Figure 14: Average lookup RPC count as a function of the fraction of the CFS servers that fail. There are 1,000 servers before the failures. Each data point is the average of 5 experiments; the error bars indicate the minimum and maximum results.

Each block has six replicas (including the main copy stored at the direct successor). After the insertions, a fraction of the servers fail without warning. Before Chord starts rebuilding its routing tables, 1,000 fetches of randomly selected blocks are attempted from a single server. Figure 13 shows the fraction of lookups that fail, and Figure 14 shows the average RPC count of the lookups.

No lookups fail when fewer than 20% of the servers fail, and very few when less than 35% fail. The reason for this is that server finger tables and successor lists provide many potential paths to carry a query around the Chord ID ring; if the most desirable finger table entry points to a failed server, CFS uses an entry that points less far around the ring. Lookups start to fail when enough servers fail that some blocks lose all six copies. For example, when 50% of the servers fail, the probability of losing all of a block's replicas is $0.5^6 = 0.016$; this is close to the value 0.013 shown in Figure 13. All of the lookup failures encountered in this experiment are due to all of a block's replicas failing; CFS was always able to find a copy of a block if one was available.

Figure 14 shows that lookups take about one RPC longer as a result of 50% of the servers failing. The RPC counts do not include attempts to contact failed servers. Lookups take longer after failures because some of the finger table entries required for fast lookups point to failed servers. If half of the finger table entries are not valid, then each RPC makes about half as much progress as expected; but one extra RPC fully corrects this.

Figure 15 shows the number of attempts to contact failed servers that occur per lookup, averaged over 1,000 block lookups. After the first time a server decides (by a timeout) that it has used a finger table or successor-list entry that points to a failed server, it does not use that server again until it has been stabilized. Given that massive failures have little effect on the availability of data or the number of RPCs per lookup, users are likely to perceive such failures because of RPC timeouts during lookups. However, Figure 15 shows that a typical block lookup shortly after a failure can expect less than one

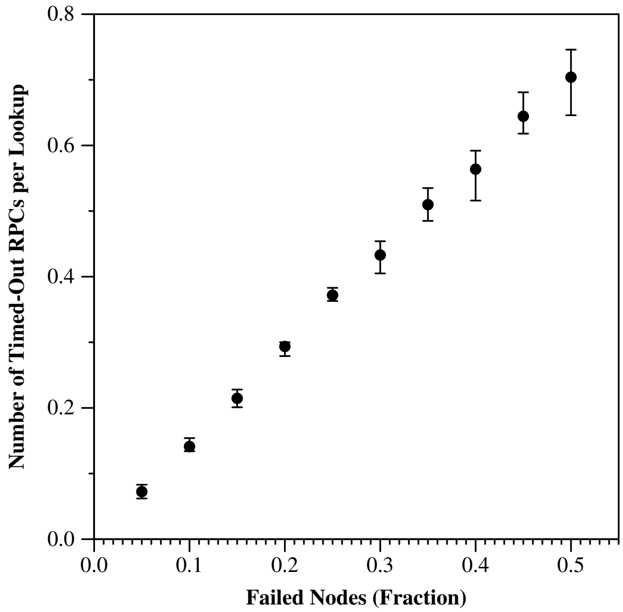

Figure 15: Number of RPC timeouts incurred during 1,000 lookups, as a function of the fraction of the CFS servers that fail. There are 1,000 servers before the failures. Each data point is the average of 5 experiments; the error bars indicate the minimum and maximum results.

timeout on the way to retrieving the desired block.

These experiments demonstrate that a large fraction of CFS servers can fail without significantly affecting data availability or performance.

8. Future Research

CFS could benefit from a keyword search system. One way to provide this would be to adopt an existing centralized search engine. A more ambitious approach would be to store the required index files using CFS itself, an idea we are pursuing.

While CFS is robust in the case of fail-stop failures, it does not specifically defend against malicious participants. A group of malicious nodes could form an internally consistent CFS system; such a system could not falsify documents, since CFS can verify authenticity, but it could incorrectly claim that a document did not exist. Future versions of CFS will defend against this by periodically checking the consistency of Chord tables with randomly chosen other nodes.

CFS must copy blocks between servers whenever a node joins or leaves the system in order to maintain the desired level of replication. If nodes often join the system for a short time before leaving, these copies will be wasted. CFS should allow for lazy replica copying.

CFS does not work through a NAT. Special arrangements might make this possible, such as a pool of global servers acting as proxies for hosts behind NATs [21].

The current CFS client uses a fixed pre-fetch window. No one size will give best performance in all situations; the right size depends on round trip time and available network bandwidth. The pre-fetch window serves a purpose similar to TCP's congestion window, and should use analogous adaptive algorithms.

9. Conclusions

CFS is a highly scalable, available and secure read-only file system. It presents stored data to applications through an ordinary file-system interface. Servers store uninterpreted blocks of data with unique identifiers. Clients retrieve blocks from the servers and interpret them as file systems.

CFS uses the peer-to-peer Chord lookup protocol to map blocks to servers. This mapping is dynamic and implicit. As a result, there is no directory information to be updated when the underlying network changes. This makes CFS both robust and scalable. CFS uses replication and caching to achieve availability and load balance. CFS replicates a block along consecutive servers in the identifier space. It caches a block along the lookup path starting to the block's server. Finally, CFS provides simple but effective protection against a single attacker inserting large amounts of data.

A prototype implementation of CFS has been implemented and evaluated on a controlled Internet-wide test-bed. Future operational deployment will likely uncover opportunities for improvement, but the current results indicate that CFS is a viable large-scale peer-to-peer system.

Acknowledgments

We are grateful to Hari Balakrishnan, John Zahorjan, and the anonymous reviewers for helpful comments, and to David Andersen for managing the RON testbed and letting us use it.

References

[1] Akamai Technologies, Inc. http://www.akamai.com/, 2001. Cambridge, MA.

[2] ANDERSEN, D., BALAKRISHNAN, H., KAASHOEK, M. F., AND MORRIS, R. Resilient overlay networks. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (Oct. 2001).

[3] CHANKHUNTHOD, A., DANZIG, P., NEERDAELS, C., SCHWARTZ, M., AND WORRELL, K. A hierarchical Internet object cache. In *Proc. Usenix Technical Conference* (Jan. 1996), pp. 153–163.

[4] CHAUM, D. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM 24*, 2 (Feb. 1981), 84–88.

[5] CLARKE, I. A distributed decentralised information storage and retrieval system. Master's thesis, University of Edinburgh, 1999.

[6] CLARKE, I., SANDBERG, O., WILEY, B., AND HONG, T. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability* (July 2000), pp. 46–66.

[7] DINGLEDINE, R., FREEDMAN, M., AND MOLNAR, D. The Free Haven project: Distributed anonymous storage service. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability* (July 2000), pp. 67–95.

[8] FAN, L., CAO, P., ALMEIDA, J., AND BRODER, A. Summary cache: a scalable wide-area web-cache sharing protocol. Tech. Rep. 1361, Computer Science Department, University of Wisconsin, Madison, Feb. 1998.

[9] FU, K., KAASHOEK, M. F., AND MAZIÈRES, D. Fast and secure distributed read-only file system. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (October 2000), pp. 181–196.

[10] GADDE, S., CHASE, J., AND RABINOVICH, M. A taste of crispy squid. In *Workshop on Internet Server Performance* (June 1998), pp. 129–136.

[11] Gnutella website. http://gnutella.wego.com.

[12] KARGER, D., LEHMAN, E., LEIGHTON, T., LEVINE, M., LEWIN, D., AND PANIGRAHY, R. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing* (May 1997), pp. 654–663.

[13] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S.,

EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. Oceanstore: An architecture for global-scale persistent storage. In *Proceeedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (November 2000), pp. 190–201.

[14] LEWIN, D. Consistent hashing and random trees: Algorithms for caching in distributed networks. Master's thesis, MIT, 1998.

[15] MALPANI, R., LORCH, J., AND BERGER, D. Making world wide web caching servers cooperate. In *Fourth International World Wide Web Conference* (1995), pp. 107–110.

[16] MAZIÈRES, D. A toolkit for user-level file systems. In *Proc. Usenix Technical Conference* (June 2001), pp. 261–274.

[17] MAZIÈRES, D., KAMINSKY, M., KAASHOEK, M. F., AND WITCHEL, E. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)* (Dec. 1999), pp. 124–139.

[18] MERKLE, R. C. A digital signature based on a conventional encryption function. In *Advances in Cryptology—CRYPTO '87* (Berlin, 1987), C. Pomerance, Ed., vol. 293 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 369–378.

[19] Mojo nation documentation. http://www.mojonation.net/docs/.

[20] Napster. http://www.napster.com.

[21] NG, T. S. E., STOICA, I., AND ZHANG, H. A waypoint service approach to connect heterogeneous internet address spaces. In *Proc. Usenix Technical Conference* (June 2001), pp. 319–332.

[22] Ohaha. `http://www.ohaha.com/design.html`, as of June 17, 2001, the Ohaha application is no longer available.

[23] ORAM, A., Ed. *Peer-to-Peer: Harnessing the Power of Disruptive Computation*. O'Reilly & Associates, 2001.

[24] PLAXTON, C., RAJARAMAN, R., AND RICHA, A. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the ACM SPAA* (June 1997), pp. 311–320.

[25] RABIN, M. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM 36*, 2 (1989), 335–348.

[26] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A scalable content-addressable network. In *Proc. ACM SIGCOMM* (San Diego, 2001).

[27] REITER, M., AND RUBIN, A. Crowds: Anonymity for web transactions. *ACM Transactions on Information and System Security 1*, 1 (Nov. 1998), 66–92.

[28] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)* (Nov. 2001).

[29] ROWSTRON, A., AND DRUSCHEL, P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (Oct. 2001).

[30] SHERMAN, A., KARGER, D., BERKHEIMER, A., BOGSTAD, B., DHANIDINA, R., IWAMOTO, K., KIM, B., MATKINS, L., AND YERUSHALMI, Y. Web caching with consistent hashing. *Computer Networks 31*, 11–16 (May 1999), 1203–1213.

[31] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. ACM SIGCOMM* (San Diego, 2001).

[32] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. Tech. Rep. TR-819, MIT, Cambridge, MA, March 2001.

[33] TYAN, T. A case study of server selection. Master's thesis, MIT, Sept. 2001.

[34] WALDMAN, M., RUBIN, A., AND CRANOR, L. F. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proc. 9th USENIX Security Symposium* (August 2000), pp. 59–72.

[35] ZHAO, B., KUBIATOWICZ, J., AND JOSEPH, A. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. Rep. UCB/CSD-01-1141, Computer Science Division, U. C. Berkeley, Apr. 2001.