Shark: Scaling File Servers via Cooperative Caching

Siddhartha Annapureddy, Michael J. Freedman, David Mazières
New York University
http://www.scs.cs.nyu.edu/shark/

Abstract

Network file systems offer a powerful, transparent interface for accessing remote data. Unfortunately, in current network file systems like NFS, clients fetch data from a central file server, inherently limiting the system's ability to scale to many clients. While recent distributed (peer-to-peer) systems have managed to eliminate this scalability bottleneck, they are often exceedingly complex and provide non-standard models for administration and accountability. We present Shark, a novel system that retains the best of both worlds—the scalability of distributed systems with the simplicity of central servers.

Shark is a distributed file system designed for large-scale, wide-area deployment, while also providing a drop-in replacement for local-area file systems. Shark introduces a novel cooperative-caching mechanism, in which mutually-distrustful clients can exploit each others' file caches to reduce load on an origin file server. Using a distributed index, Shark clients find nearby copies of data, even when files originate from different servers. Performance results show that Shark can greatly reduce server load and improve client latency for read-heavy workloads both in the wide and local areas, while still remaining competitive for single clients in the local area. Thus, Shark enables modestly-provisioned file servers to scale to hundreds of read-mostly clients while retaining traditional usability, consistency, security, and accountability.

1 Introduction

Users of distributed computing environments often launch similar processes on hundreds of machines nearly simultaneously. Running jobs in such an environment can be significantly more complicated, both because of data-staging concerns and the increased difficulty of debugging. Batch-oriented tools, such as Condor [9], can provide I/O transparency to help distribute CPU-intensive applications. However, these tools are ill-suited to tasks like distributed web hosting and network measurement, in which software needs low-level control of network functions and resource allocation. An alternative is frequently seen on network test-beds such as RON [2] and PlanetLab [24]: users replicate their programs, along with some minimal execution environment, on every machine before launching a distributed application.

Replicating execution environments has a number of drawbacks. First, it wastes resources, particularly bandwidth. Popular file synchronization tools do not optimize for network locality, and they can push many copies of the same file across slow network links. Moreover, in a shared environment, multiple users will inevitably copy the exact same files, such as popular OS add-on packages with language interpreters or shared libraries. Second, replicating run-time environments requires hard state, a scarce resource in a shared test-bed. Programs need sufficient disk space, yet idle environments continue to consume disk space, in part because the owners are loathe to consume the bandwidth and effort required for redistribution. Third, replicated run-time environments differ significantly from an application's development environment, in part to conserve bandwidth and disk space. For instance, users usually distribute only stripped binaries, not source or development tools, making it difficult to debug running processes in a distributed system.

Shark is a network file system specifically designed to support widely distributed applications. Rather than manually replicate program files, users can place a distributed application and its entire run-time environment in an exported file system, and simply execute the program directly from the file system on all nodes. In a chrooted environment such as PlanetLab, users can even make `/usr/local` a symbolic link to a Shark file system, thereby trivially making all local software available on all test-bed machines.

Of course, the big challenge faced by Shark is scalability. With a normal network file system, if hundreds of clients suddenly execute a large, 40MB C++ program from a file server, the server quickly saturates its network uplink and delivers unacceptable performance. Shark, however, scales to large numbers of clients through a locality-aware cooperative cache. When reading an uncached file, a Shark client avoids transferring the file or even chunks of the file from the server, if the same data can be fetched from another, preferably nearby, client. For world-readable files, clients will even download nearby cached copies of identical files—or even file chunks—

originating from different servers.

Shark leverages a locality-aware, peer-to-peer distributed index [10] to coordinate client caching. Shark clients form self-organizing clusters of well-connected machines. When multiple clients attempt to read identical data, these clients locate nearby replicas and stripe downloads from each other in parallel. Thus, even modestly-provisioned file servers can scale to hundreds, possibly thousands, of clients making mostly read accesses.

There have been serverless, peer-to-peer file systems capable of scaling to large numbers of clients, notably Ivy [23]. Unfortunately, these systems have highly non-standard models for administration, accountability, and consistency. For example, Ivy spreads hard state over multiple machines, chosen based on file system data structure hashes. This leaves no single entity ultimately responsible for the persistence of a given file. Moreover, peer-to-peer file systems are typically noticeably slower than conventional network file systems. Thus, in both accountability and performance they do not provide a substitute for conventional file systems. Shark, by contrast, exports a traditional file-system interface, is compatible with existing backup and restore procedures, provides competitive performance on the local area network, and yet easily scales to many clients in the wide area.

For workloads with no read sharing between users, Shark offers performance that is competitive with traditional network file systems. However, for shared read-heavy workloads in the wide area, Shark greatly reduces server load and improves client latency. Compared to both NFSv3 [6] and SFS [21], a secure network file system, Shark can reduce server bandwidth usage by nearly an order of magnitude and can provide a 4x-6x improvement in client latency for reading large files, as shown by both local-area experiments on the Emulab [36] test-bed and wide-area experiments on the PlanetLab [24] test-bed.

By providing scalability, efficiency, and security, Shark enables network file systems to be employed in environments where they were previously impractical. Yet Shark retains their attractive API, semantics, and portability: Shark interacts with the local host using an existing network file system protocol (NFSv3) and runs in user space.

The remainder of this paper in organized as follows. Section 2 details the design of Shark: its file-system components, caching and security protocols, and distributed index operations. Section 3 describes its implementation, and Section 4 evaluates Shark's performance. Section 5 discusses related work, and Section 6 concludes.

2 Shark Design

Shark's design incorporates a number of key ideas aimed at reducing the load on the server and improving client-perceived latencies. Shark enables clients to securely

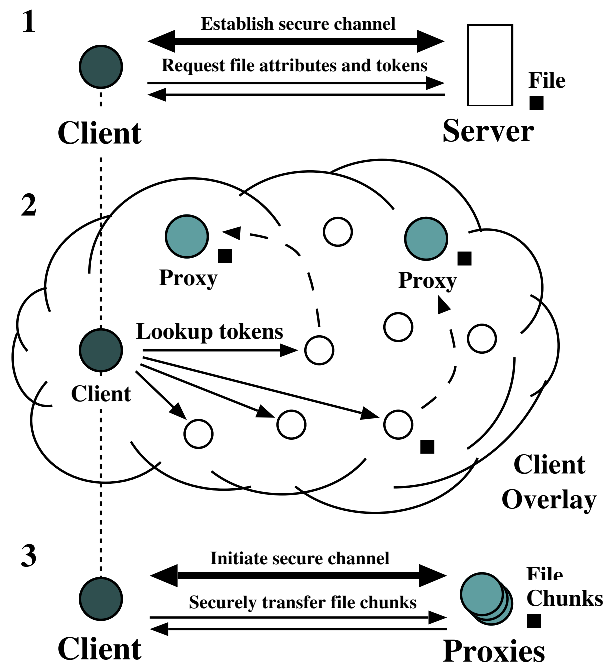

Figure 1: *Shark System Overview.* A client machine simultaneously acts as a client (to handle local application file system accesses), as a proxy (to serve cached data to other clients), and as a node (within the distributed index overlay). In a real deployment, there may be multiple file servers that each host separate file systems, and each client may access multiple file systems. For simplicity, however, we show a single file server.

mount remote file systems and efficiently access them. When a client is the first to read a particular file, it fetches the data from the file server. Upon retrieving the file, the client caches it and registers itself as a *replica proxy* (or *proxy* for short) for the "chunks" of the file in the distributed index. Subsequently, when another client attempts to access the file, it discovers proxies for the file chunks by querying the distributed index. The client then establishes a secure channel to multiple such proxies and downloads the file chunks in parallel. (Note that the client and the proxy are mutually distrustful.) Upon fetching these chunks, the client registers itself also as a proxy for these chunks.

Figure 1 provides an overview of the Shark system. When a client attempts to read a file, it queries the file server for the file's attributes and some opaque tokens (Step 1 as shown). One token identifies the contents of the whole file, while other tokens each identify a particular *chunk* of the file. A Shark server divides a file into chunks by running a Rabin fingerprint algorithm on the file [22]. This technique splits a file along specially chosen boundaries in such a way that preserves data commonalities across files, for example, between file versions or when concatenating files, such as building program libraries from object files.

Next, a client attempts to discover replica proxies for the particular file via Shark's distributed index (Step 2). Shark clients organize themselves into a key/value indexing infrastructure, built atop a peer-to-peer structured routing overlay [10]. For now, we can visualize this layer as exposing two operations, put and get: A client executes put to declare that it has something; get returns the list of clients who have something. A Shark client uses its tokens to derive *indexing keys* that serve as inputs to these operations. It uses this distributed index to register itself and to find other nearby proxies caching a file chunk.

Finally, a client connects to several of these proxies, and it requests various chunks of data from each proxy in parallel (Step 3). Note, however, that clients themselves are mutually distrustful, so Shark must provide various mechanisms to guarantee secure data sharing: (1) Data should be encrypted to preserve confidentiality and should be decrypted only by those with appropriate read permissions. (2) A malicious proxy should not be able to break data integrity by modifying content without a client detecting the change. (3) A client should not be able to download large amounts of even encrypted data without proper read authorization.

Shark uses the opaque tokens generated by the file server in several ways to handle these security issues. (1) The tokens serve as a shared secret (between client and proxy) with which to derive symmetric cryptographic keys for transmitting data from proxy to client. (2) The client can verify the integrity of retrieved data, as the token acts to bind the file contents to a specific verifiable value. (3) A client can "prove" knowledge of the token to a proxy and thus establish read permissions for the file. Note that the indexing keys used as input to the distributed index are only derived from the token; they do not in fact expose the token's value or otherwise destroy its usefulness as a shared secret.

Shark allows clients to share common data segments on a sub-file granularity. As a file server provides the tokens naming individual file chunks, clients can share data at the granularity of chunks as opposed to whole files.

In fact, Shark provides *cross-file-system sharing* when tokens are derived solely from file contents. Consider the case when users attempt to mount `/usr/local` (for the same operating system) using different file servers. Most of the files in these directories are identical and even when the file versions are different, many of the chunks are identical. Thus, even when distinct subsets of clients access different file servers to retrieve tokens, one can still act as a proxy for the other to transmit the data.

In this section, we first describe the Shark file server (Section 2.1), then discuss the file consistency provided by Shark (2.2). Section 2.3 describes Shark's cooperative caching, its cryptographic operations, and client-proxy protocols. Finally, we present Shark's chunking algorithm (2.4) and its distributed index (2.5) in more depth.

2.1 Shark file servers

Shark names file systems using self-certifying pathnames, as in SFS [21]. These pathnames *explicitly* specify all information necessary to securely communicate with remote servers. Every Shark file system is accessible under a pathname of the form:

$$/\texttt{shark}/@server, pubkey$$

A Shark server exports local file systems to remote clients by acting as an NFS loop-back client. A Shark client provides access to a remote file system by automounting requested directories [21]. This allows a client-side Shark NFS loop-back server to provide unmodified applications with seamless access to remote Shark file systems. Unlike NFS, however, all communication with the file server is sent over a secure channel, as the self-certifying pathname includes sufficient information to establish a secure channel.

System administrators manage a Shark server identically to an NFS server. They can perform backups, manage access controls with little difference. They can configure the machine to taste, enforce various policies, perform security audits etc. with *e*xisting tools. Thus, Shark provides system administrators with a familiar environment and thus can be deployed painlessly.

2.2 File consistency

Shark uses two network file system techniques to improve read performance and decrease server load: leases [11] and AFS-style whole-file caching [14]. When a user attempts to read any portion of a file, the client first checks its disk cache. If the file is not already cached or the cached copy is not up to date, the client fetches a new version from Shark (either from the cooperative cache or directly from the file server).

Whenever a client makes a read RPC to the file server, it gets a read lease on that particular file. This lease corresponds to a commitment from the server to notify the client of any modifications to the file within the lease's duration. Shark uses a default lease duration of five minutes. Thus, if a user attempts to reads from a file—and if the file is cached, its lease is not expired, and no server notification (or *callback*) has been received—the read succeeds immediately using the cached copy.

If the lease has already expired when the user attempts to read the file, the client contacts the file server for fresh

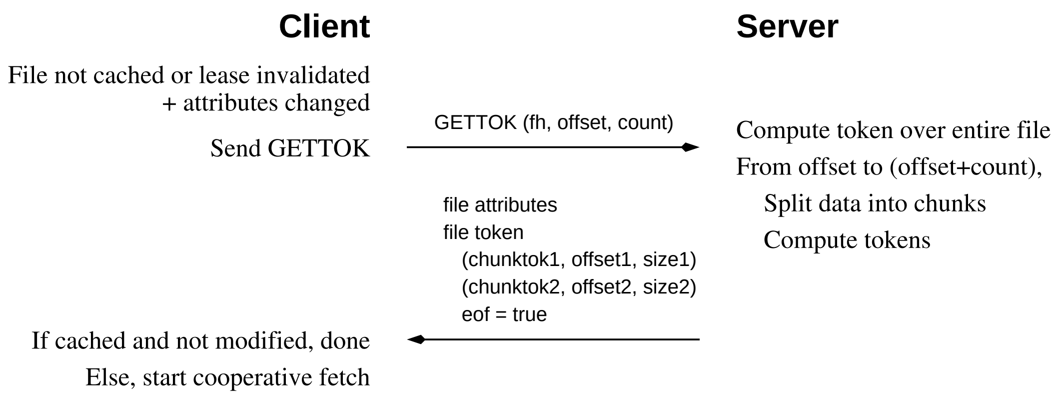

Figure 2: Shark GETTOK RPC

file attributes. The attributes, which include file permissions, mode, size, etc., also provide the file's modification and inode change times. If these times are the same as the cached copy, no further action is necessary: the cached copy is fresh and the client renews its lease. Otherwise, the client needs to fetch a new version from Shark.

While these techniques reduce unnecessary data transfers when files have not been modified, each client needs to refetch the entire file after any modification from the server. Thus, large numbers of clients for a particular file system may overload the server and offer poor performance. Two techniques alleviate the problem: Shark fetches only modified chunks of a file, while its cooperative caching allows clients to fetch data from each other instead of from the server.

While Shark attempts to handle reads within its cooperative cache, all writes are sent to the origin server. When any type of modification occurs, the server must invalidate all unexpired leases, update file attributes, recompute its file token, and update its chunk tokens and boundaries.

We note that a reader can get a mix of old and new file data if a file is modified *while* the reader is fetching file attributes and tokens from the server. (This condition can occur when fetching file tokens requires multiple RPCs, as described next.) However, this behavior is no different from NFS, but it could be changes using AFS-style whole-file overwrites [14].

2.3 Cooperative caching

File reads in Shark make use of one RPC procedure not in the NFS protocol, GETTOK, as shown in Figure 2.

GETTOK supplies a file handle, offset, and count as arguments, just as in a READ RPC. However, instead of returning the actual file data, it returns the file's attributes, the *file token*, and a vector of *chunk descriptions*. Each chunk description identifies a specific extent of the file by offset and size, and includes a *chunk token* for that extent. The server will only return up to 1,024 chunk descriptions in one GETTOK call; the client must issue multiple calls for larger files.

The file attributes returned by GETTOK include sufficient information to determine if a local cached copy is up-to-date (as discussed). The tokens allow a client (1) to discover current proxies for the data, (2) to demonstrate read permission for the data to proxies, and (3) to verify the integrity of data retrieved from proxies. First, let us specify how Shark's various tokens and keys are derived.

Content-based naming. Shark names content with cryptographic hash operations, as given in Table 1.

A *file token* is a 160-bit value generated by a cryptographic hash of the file's contents F and some optional per-file randomness r that a server may use as a key for each file (discussed later):

$$T_F = tok(F) = \mathsf{HMAC}_r(F)$$

Throughout our design, HMAC is a keyed hash function [4], which we instantiate with SHA-1. We assume that SHA-1 acts as a collision-resistant hash function, which implies that an adversary cannot find an alternate input pair that yields the same T_F.[1]

The *chunk token* T_{F_i} in a chunk description is also computed in the same manner, but only uses the particular chunk of data (and optional randomness) as an input to SHA-1, instead of the entire file F. As file and chunk tokens play similar roles in the system, we use T to refer to either type of token indiscriminately.

The *indexing key* I used in Shark's distributed index is simply computed by $\mathsf{HMAC}_T(\mathsf{I})$. We key the HMAC function with T and include a special character I to signify indexing. More specifically, I_F refers to the indexing key for file F, and I_{F_i} for chunk F_i.

The use of such server-selected randomness r ensures that an adversary cannot guess file contents, given only

[1] While our current implementation uses SHA-1, we could similarly instantiate HMAC with SHA-256 for greater security.

Symbol	Description	Generated by ...	Only known by ...
F	File		Server and approved readers
F_i	ith file chunk	Chunking algorithm	Parties with access to F
r	Server-specific randomness	$r = \mathsf{PRNG}()$ or $r = 0$	Parties with access to F
T	File/chunk token	$tok(F) = \mathsf{HMAC}_r(F)$	Parties with access to F/F_i
$\mathsf{I}, \mathsf{E}, \mathsf{A}_C, \mathsf{A}_P$	Special constants	System-wide parameters	Public
I	Indexing key	$\mathsf{HMAC}_T(\mathsf{I})$	Public
r_C, r_P	Session nonces	$r_C, r_P = \mathsf{PRNG}()$	Parties exchanging F/F_i
$Auth_C$	Client authentication token	$\mathsf{HMAC}_T(\mathsf{A}_C, C, P, r_C, r_P)$	Parties exchanging F/F_i
$Auth_P$	Proxy authentication token	$\mathsf{HMAC}_T(\mathsf{A}_P, P, P, r_P, r_C)$	Parties exchanging F/F_i
K_E	Encryption key	$\mathsf{HMAC}_T(\mathsf{E}, C, P, r_C, r_P)$	Parties exchanging F/F_i

Table 1: Notation used for Shark values

I. Otherwise, if the file is small or stylized, an adversary may be able to perform an offline brute-force attack by enumerating all possibilities.

On the flip-side, omitting this randomness enables cross-file-system sharing, as its content-based naming can be made independent of the file server. That is, when r is omitted and replaced by a string of 0s, the distributed indexing key is dependent only on the contents of F: $I_F = \mathsf{HMAC}_{\mathsf{HMAC}_0(F)}(\mathsf{I})$. Cross-file-system sharing can improve client performance and server scalability when nearby clients use different servers. Thus, the system allows one to trade-off additional security guarantees with potential performance improvements. By default, we omit this randomness for world-readable files, although configuration options can override this behavior.

The cooperative-caching read protocol. We now specify in detail the cooperative-caching protocol used by Shark. The main goals of the protocol are to reduce the load on the server and to improve client-perceived latencies. To this end, a client tries to download chunks of a file from multiple proxies in parallel. At a high level, a client first fetches the tokens for the chunks that comprise a file. It then contacts nearby proxies holding each chunk (if such proxies exists) and downloads them accordingly. If no other proxy is caching a particular chunk of interest, the client falls back on the server for that chunk.

The client sends a GETTOK RPC to the server and fetches the whole-file token, the chunk tokens, and the file's attributes. It then checks its cache to determine whether it has a fresh local copy of the file. If not, the client runs the following cooperative read protocol.

The client always attempts to fetch k chunks in parallel. We can visualize the client as spawning k threads, with each thread responsible for fetching its assigned chunk.[2] Each thread is assigned a *random* chunk F_i from the list of needed chunks. The thread attempts to discover nearby proxies caching that chunk by querying the distributed index using the primitive $get(I_{F_i} = \mathsf{HMAC}_{T_{F_i}}(\mathsf{I}))$. If this *get* request fails to find a proxy or does not find one within a specified time, the client fetches the chunk from the server. After downloading the entire chunk, the client announces itself in the distributed index as a proxy for F_i.

If the get request returns several proxies for chunk F_i, the client chooses one with minimal latency and establishes a secure channel with the proxy, as described later. If the security protocol fails (perhaps due to a malicious proxy), the connection to the proxy fails, or a newly specified time is exceeded, the thread chooses another proxy from which to download chunk F_i. Upon downloading F_i, the client verifies its integrity by checking whether $T_{F_i} \stackrel{?}{=} tok(F_i)$. If the client fails to successfully download F_i from any proxy after a fixed number of attempts, it falls back onto the origin file server.

Reusing proxy connections. While a client is downloading a chunk from a proxy, it attempts to reuse the connection to the proxy by *negotiating* for other chunks. The client picks α random chunks still needed. It computes the corresponding α indexing keys and sends these to the proxy. The proxy responds with those γ chunks, among the α requested, that it already has. If $\gamma = 0$, the proxy responds instead with β keys corresponding to chunks that it does have. The client, upon downloading the current chunk, selects a new chunk from among those negotiated (*i.e.*, needed by the client and known to the proxy). The client then proves read permissions on the new chunk and begins fetching the new chunk. If no such chunks can be negotiated, the client terminates the connection.

Client-proxy interactions. We now describe the secure communication mechanisms between clients and proxies that ensure confidentiality and authorization. We already described how clients achieve data integrity by verifying the contents of files/chunks by their tokens.

To prevent adversaries from passively reading or actively modifying content while in transmission, the client and proxy first derive a symmetric encryption key K_E be-

[2] Our implementation is structured using asynchronous events and callbacks within a single process, we use the term "thread" here only for clarity of explanation.

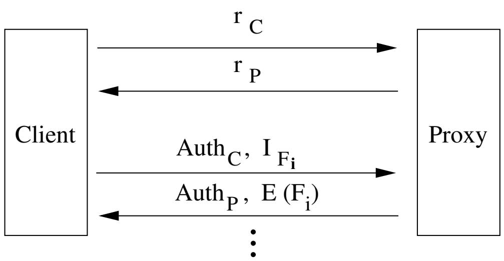

Figure 3: Shark session establishment protocol

fore transmitting a chunk. As the token T_{F_i} already serves as a shared secret for chunk F_i, the parties can simply use it to generate this key.

Figure 3 shows the protocol by which Shark clients establish a secure session. First, the parties exchange fresh, random 20-byte nonces r_C and r_P upon initiating a connection. For each chunk to be sent over the connection, the client must signal the proxy which token T_{F_i} to use, but it can do so without exposing information to eavesdroppers or malicious proxies by simply sending I_{F_i} in the clear. Using these nonces and knowledge of T_{F_i}, each party computes authentication tokens as follows:

$$\begin{aligned} Auth_C &= \mathsf{HMAC}_{T_{F_i}}(\mathsf{A}_C, C, P, r_C, r_P) \\ Auth_P &= \mathsf{HMAC}_{T_{F_i}}(\mathsf{A}_P, P, C, r_P, r_C) \end{aligned}$$

The $Auth_C$ token proves to the proxy that the client actually has the corresponding chunk token T_{F_i} and thus read permissions on the chunk. Upon verifying $Auth_C$, the proxy replies with $Auth_P$ and the chunk F_i after applying E to it.

In our current implementation, E is instantiated by a symmetric block encryption function, followed by a MAC covering the ciphertext. However, we note that $Auth_P$ already serves as a MAC for the *content*, and thus this additional MAC is not strictly needed. [3] The symmetric encryption key K_E for E is derived in a similar manner as before:

$$K_E = \mathsf{HMAC}_{T_{F_i}}(\mathsf{E}, C, P, r_C, r_P)$$

An additional MAC key can be similarly derived by replacing the special character E with M. Shark's use of fresh nonces ensure that these derived authentication tokens and keys cannot be replayed for subsequent requests.

Upon deriving this symmetric key K_E, the proxy encrypts the data within a chunk using 128-bit AES in counter mode (AES-CTR). Per each 16-byte AES block, we use the block's offset within the chunk/file as its counter.

The proxy protocol has READ and READDIR RPCs similar to NFS, except they specify the indexing key I and $Auth_C$ to name a file (which is server independent), in place of a file handle. Thus, after establishing a connection, the client begins issuing read RPCs to the proxy; the client decrypts any data it receives in response using K_E and the proper counter (offset).

While this block encryption prevents a client without T_{F_i} from decrypting the data, one may be concerned if some unauthorized client can download a large number of encrypted blocks, with the hope of either learning K_E later or performing some offline attack. The proxy's explicit check of $Auth_C$ prevents this. Similarly, the verifiable $Auth_P$ prevents a malicious party that does not hold F_i from registering itself under the public I_{F_i} and then wasting the client's bandwidth by sending invalid blocks (that later will fail hash verification).

Thus, Shark provides strong data integrity guarantees to the client and authorization guarantees to the proxy, even in the face of malicious participants.

2.4 Exploiting file commonalities

We describe the chunking method by which Shark can leverage file commonalities. This method (used by LBFS [22]) avoids a sensitivity to file-length changes by setting chunk boundaries, or *breakpoints*, based on file contents, rather than on offset position. If breakpoints were selected only by offset—for instance, by breaking a file into aligned 16KB chunks—a single byte added to the front of a file would change all breakpoints and thus all chunk tokens.

To divide a file into chunks, we examine every overlapping 48-byte region, and if the low-order 14 bits of the region's Rabin fingerprint [25] equals some globally-chosen value, the region constitutes a breakpoint. Assuming random data, the expected chunk size is therefore $2^{14} = 16$KB. To prevent pathological cases (such as long strings of 0), the algorithm uses a minimum chunk size of 2KB and maximum size of 64KB. Therefore, modifications within a chunk will minimize changes to the breakpoints: either only the chunk will change, one chunk will split into two, or two chunks will merge into one.

Content-based chunking enables Shark to exploit file commonalities: Even if proxies were reading different versions of the same file or different files altogether, a client can discover and download common data chunks, as long as they share the same chunk token (and no server-specific randomness). As the fingerprint value is global, this chunking commonality also persists across multiple file systems.

[3] The results of Krawczyk [15] speaking on the *generic* security concerns of "authenticate-and-encrypt" are not really relevant here, as we already expose the raw output of our MAC via I_{F_i} and thus implicitly assume that HMAC does not leak any information about its contents. Thus, the inclusion of $Auth_P$ does not introduce any *additional* data confidentiality concerns.

2.5 Distributed indexing

Shark seeks to enable data sharing both between files on the same file system that contain identical data chunks across different file systems. This functionality is not supported by the simple server-based approach of indexing clients, whereby the file server stores and returns information on which clients are caching which chunks. Thus, we use a *global* distributed index for all Shark clients, even those accessing different Shark file systems.

Shark uses a structured routing overlay [33, 26, 29, 37, 19] to build its distributed index. The system maps opaque keys onto nodes by hashing their value onto a semantic-free identifier (ID) space; nodes are assigned identifiers in the same ID space. It allows scalable key lookup (in $O(\log(n))$ overlay hops for n-node systems), reorganizes itself upon network membership changes, and provides robust behavior against failure.

While many routing overlays optimize routes along the underlay, most are designed as part of distributed hash tables to store immutable data. In contrast, Shark stores only small references about which clients are caching what data: It seeks to allow clients to locate *copies* of data, not merely to find network efficient *routes* through the overlay. In order to achieve such functionality, Shark uses Coral [10] as its distributed index.

System overview. Coral exposes two main protocols: put and get. A Shark client executes the get protocol with its indexing key I as input; the protocol returns a list of proxy addresses that corresponds to some *subset* of the unexpired addresses put under I, taking locality into consideration. put takes as input I, a proxy's address, and some expiry time.

Coral provides a *distributed sloppy hash table* (DSHT) abstraction, which offers weaker consistency than traditional DHTs. It is designed for soft-state where multiple values may be stored under the same key. This consistency is well-suited for Shark: A client need not discover *all* proxies for a particular file, it only needs to find *several*, *nearby* proxies.

Coral caches key/value pairs at nodes whose IDs are close (in terms of identifier space distance) to the key being referenced. To lookup the client addresses associated with a key I, a node simply traverses the ID space with RPCs and, as soon as it finds a remote peer storing I, it returns the corresponding list of values. To insert a key/value pair, Coral performs a two-phase operation. In the "forward" phase, Coral routes to nodes successively closer to I and stops when happening upon a node that is both *full* (meaning it has reached the maximum number of values for the key) and *loaded* (which occurs when there is heavy write traffic for a particular key). During the "reverse" phase, the client node attempts to insert the value at the closest node seen. See [10] for more details.

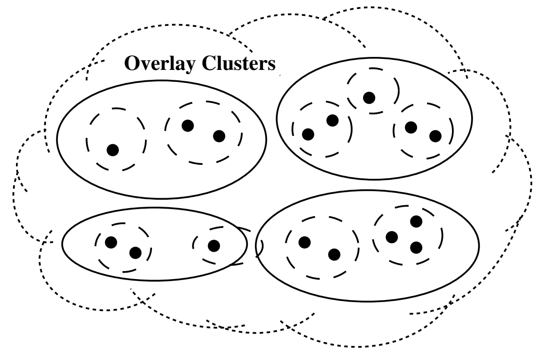

Figure 4: *Coral's three-level hierarchical overlay structure.* Nodes (solid circles) initially query others in their same high-level clusters (dashed rings), whose pointers reference other proxies caching the data within the same small-diameter cluster. If a node finds such a mapping to a replica proxy in the highest-level cluster, the get finishes. Otherwise, it continues among farther, lower-level nodes (solid rings), and finally, if need be, to any node within the system (the cloud).

To improve locality, these routing operations are not initially performed across the entire global overlay: Each Coral node belongs to several distinct routing structures called *clusters*. Each cluster is characterized by a maximum desired network round-trip-time (RTT) called the *diameter*. The system is parameterized by a fixed hierarchy of diameters, or *levels*. Every node belongs to one cluster at each level, as shown in Figure 4. Coral queries nodes in fast clusters before those in slower clusters. This both reduces the latency of lookups and increases the chances of returning values stored by nearby nodes.

Handle concurrency via "atomic" put/get. Ideally, Shark clients should fetch each file chunk from a Shark server only once. However, a DHT-like interface which exposes two methods, put and get, is not sufficient to achieve this behavior. For example, if clients were to wait until completely fetching a file before referencing themselves, other clients simultaneously downloading the file will start transferring file contents from the server. Shark mitigates this problem by using Coral to request *chunks*, as opposed to whole files: A client delays its announcement for only the time needed to fetch a chunk.

Still, given that Shark is designed for environments that may experience abrupt flash crowds—such as when test-bed or grid researchers fire off experiments on hundreds of nodes almost simultaneously and reference large executables or data files when doing so—we investigated the practice of clients optimistically inserting a mapping to themselves upon initiating a request. A production use of Coral in a web-content distribution network takes a simi-

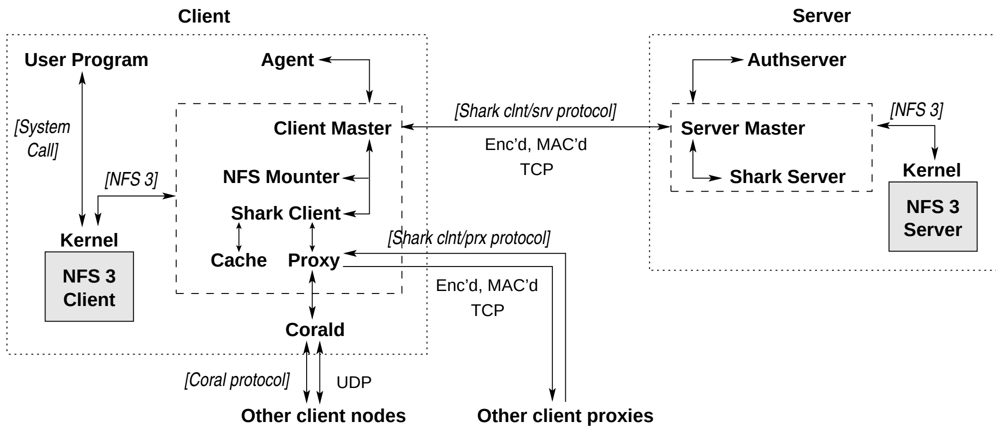

Figure 5: The Shark system components

lar approach when fetching whole web objects [10].

Even using this approach, we found that an origin server can see redundant downloads of the same file when initial requests for a newly-popular file occur synchronously. We can imagine this condition occurring in Shark when users attempt to simultaneously install software on all test-bed hosts.

Such redundant fetches occur under the following race condition: Consider that a mapping for file F (and thus I_F) is not yet inserted into the system. Two nodes both execute $get(I_F)$, then perform a put. On the node closest to I_F, the operations serialize with both gets being handling (and thus returning no values) before either put.

Simply inverting the order of operations is even worse. If multiple nodes first perform a put, followed by a get, they can discover one another and effectively form cycles waiting for one another, with nobody actually fetching the file from the server.

To eliminate this condition, we extended store operations in Coral to provide return status information (like test-and-set in shared-memory systems). Specifically, we introduce a single put/get RPC which atomically performs both operations. The RPC behaves similar to a put as described above, but also returns the first values discovered in *either* direction. (Values in the forward put direction help performance; values in the reverse direction prevent this race condition.)

While of ultimately limited use in Shark given small chunk sizes, this extension also proved beneficial for other applications seeking a distributed index abstraction [10].

3 Implementation

Shark consists of three main components, the server-side daemon `sharksd`, the client-side daemon `sharkcd` and the coral daemon `corald`, as shown in Figure 5. All three components are implemented in C++ and are built using the SFS toolkit [20]. The file-system daemons interoperate with the SFS framework, using its automounter, authentication daemon, etc. `corald` acts as a node within the Coral indexing overlay; a full description can be found in [10].

`sharksd`, the server-side daemon, is implemented as a loop-back client which communicates with the kernel NFS server. `sharksd` incorporates an extension of the NFSv3 protocol—the GETTOK RPC—to support file- and chunk-token retrieval. When `sharksd` receives a GETTOK call, it issues a series of READ calls to the kernel NFS server and computes the tokens and chunk breakpoints. It caches these tokens for future reference. `sharksd` required an additional 400 lines of code to the SFS read-write server.

`sharkcd`, the client-side daemon, forms the biggest component of Shark. In addition to handling user requests, it transparently incorporates whole-file caching and the client- and server-side functionality of the Shark cooperative cache. The code is 12,000 lines.

`sharkcd` comprises an NFS loop-back server which traps user requests and forwards them to either the origin file server or a Shark proxy. In particular, a read for a file block is intercepted by the loop-back server and translated into a series of READ calls to fetch the entire file. The cache-management subsystem of `sharkcd` stores all files that are being fetched locally on disk. This cache provides a thin wrapper around file-system calls to enforce disk usage accounting. Currently, we use the LRU mechanism to evict files from the cache. The cache names are also chosen carefully to fit in the kernel name cache.

The server side of the Shark cooperative cache implements the proxy, accepting connections from other clients. If this proxy cannot immediately satisfy a request, it registers a callback for the request, responding when the block has been fetched. The client side of the Shark cooperative cache implements the various fetching mechanism discussed in Section 2.3. For every file to be fetched, the client maintains a vector of objects representing connections to different proxies. Each object is responsible for fetching a sequence of chunks from the proxy (or a range of blocks when chunking is not being performed and nodes query only by file token).

An early version of `sharkcd` also supported the use of **xfs**, a device driver bundled with the ARLA [35] implementation of AFS, instead of NFS. However, given that the PlanetLab environment, on which we performed our testing, does not support **xfs**, we do not present those results in this paper.

During Shark's implementation, we discovered and fixed several bugs in both the OpenBSD NFS server and the **xfs** implementation.

4 Evaluation

This section evaluates Shark against NFSv3 and SFS to quantify the benefits of its cooperative-caching design for read-heavy workloads. To measure the performance of Shark against these file systems, without the gain from cooperative caching, we first present microbenchmarks for various types of file-system access tests, both in the local-area and across the wide-area. We also evaluate the efficacy of Shark's chunking mechanism in reducing redundant transfers.

Second, we measure Shark's cooperative caching mechanism by performing read tests both within the controlled Emulab LAN environment [36] and in the wide-area on the PlanetLab v3.0 test-bed [24]. In all experiments, we start with cold file caches on all clients, but first warm the server's chunk token cache. The server required 0.9 seconds to compute chunks for a 10 MB random file, and 3.6 seconds for a 40 MB random file.

We chose to evaluate Shark on Emulab, in addition to wide-area tests on PlanetLab, in order to test Shark in a more controlled, native environment: While Emulab allows one to completely reserve machines, individual PlanetLab hosts may be executing tens or hundreds of experiments (slices) simultaneously. In addition, most PlanetLab hosts implement bandwidth caps of 10 Mb/sec across all slices. For example, on a local PlanetLab machine operating at NYU, a Shark client took approximately 65 seconds to read a 40 MB file from the local (non-PlanetLab) Shark file server, while a non-PlanetLab client on the same network took 19.3 seconds. Furthermore, deployments of Shark on large LAN clusters (for example, as part of grid computing environments) may experience similar results to those we report.

The server in all the microbenchmarks and the PlanetLab experiments is a 1.40 GHz Athlon at NYU, running OpenBSD 3.6 with 512 MB of memory. It runs the corresponding server daemons for SFS and Shark. All microbenchmark and PlanetLab clients used in the experiments ran Fedora Core 2 Linux. The server used for Emulab tests was a host in the Emulab test-bed; it did not simultaneously run a client. All Emulab hosts ran Red Hat Linux 9.0.

The Shark client and server daemons interact with the respective kernel NFS modules using the *loopback* interface. On the Red Hat 9 and Fedora Core 2 machines, where we did our testing, the loopback interface has a maximum MTU of 16436 bytes and any transfer of blocks of size $>=$ 16 KB results in IP fragmentation which appears to trigger a bug in the kernel NFS code. Since we could not increase the MTU size of the loopback interface, we limited both Shark and SFS to use 8 KB blocks. NFS, on the other hand, issued UDP read requests for blocks of 32 KB over the *ethernet* interface without any problems. These settings could have affected our measurements.

4.1 Alternate cooperative protocols

This section considers several alternative cooperative-caching strategies for Shark in order to characterize the benefits of various design decisions.

First, we examine whether clients should issue requests for chunks sequentially (seq), as opposed to choosing a $random$ (previously unread) chunk to fetch. There are two additional strategies to consider when performing sequential requests: Either the client immediately pre-announces itself for a particular chunk upon requesting it (with an "atomic" put/get as in Section 2.5), or the client waits until it finishes fetching a chunk before announcing itself (via a put). We consider such sequential strategies to examine the effect of disk scheduling latency: for single clients in the local area, one intuits that the random strategy limits the throughput to that imposed by the file server's disk seek time, while we expect the network to be the bottleneck in the wide area. Yet, when multiple clients operate concurrently, one intuits that the random strategy allows all clients to fetch independent chunks from the server and later trade these chunks among themselves. Using a purely sequential strategy, the clients all advance only as fast as the few clients that initially fetch chunks from the server.

Second, we disable the $negotiation$ process by which clients may reuse connections with proxies and thus download multiple chunks once connected. In this case, the client must query the distributed index for each chunk.

4.2 Microbenchmarks

For the local-area microbenchmarks, we used a local machine at NYU as a Shark client. Maximum TCP throughput between the local client and server, as measured by

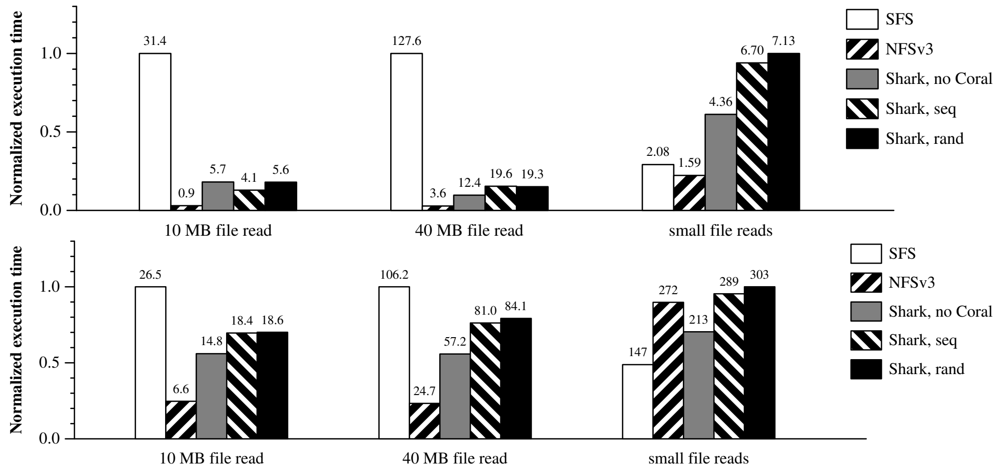

Figure 6: *Local-area* (top) *and wide-area* (bottom) *microbenchmarks.* Normalized application performance for various types of file-system access. Execution times in seconds appear above the bars.

`ttcp`, was 11.14 MB/sec. For wide-area microbenchmarks, we used a client machine located at the University of Texas at El Paso. The average round-trip-time (RTT) between this host and the server, as measured by `ping`, is 67 ms. Maximum TCP throughput was 1.07 MB/sec.

Access latency. We measure the time necessary to perform four types of file-system accesses: (1) to read 10 MB and (2) 40 MB large random files on remote hosts, and (3) to read large numbers of small files. The small file test attempts to read 1,000 1 KB files evenly distributed over ten directories.

We performed single-client microbenchmarks to measure the performance of Shark. Figure 6 shows the performance on the local- and wide-area networks for these three experiments, We compare SFS, NFS, and three Shark configurations, *viz* Shark without calls to its distributed indexing layer ($nocoral$), fetching chunks from a file sequentially (seq), and fetching chunks in random order ($rand$). Shark issues up to eight outstanding RPCs (for seq and $rand$, fetching four chunks simultaneously with two outstanding RPCs per chunk). SFS sends RPCs as requested by the NFS client in the kernel.

For all experiments, we report the normalized *median* value over three runs. We interleaved the execution of each of the five file systems over each run. We see that Shark is competitive across different file system access patterns and is optimized for large read operations.

Chunking. In this microbenchmark, we validate that Shark's chunking mechanism reduces redundant data transfers by exploiting data commonalities.

We first read the `tar` file of the entire source tree for emacs v20.6 over a Shark file system, and then read the `tar` file of the entire source tree for emacs v20.7. We note that of the 2,083 files or directories that comprise these two file archives, 1,425 have not changed between versions (*i.e.*, they have the identical `md5` sum), while 658 of these have changed.

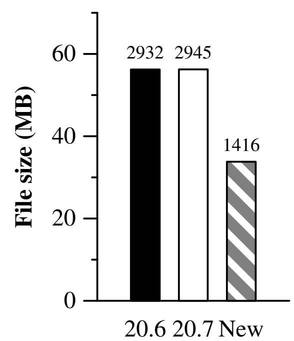

Figure 7: *Bandwidth savings from chunking.* "New" reflects the number of megabytes that need to be transferred when reading emacs 20.7 given 20.6. Number of chunks comprising each transfer appears above the bars.

Figure 7 shows the amount of bandwidth savings that the chunking mechanism provides when reading the newer emacs version. When `emacs-20.6.tar` has been cached, Shark only transfers 33.8 MB (1416 chunks) when reading `emacs-20.7.tar` (of size 56.3 MB).

4.3 Local-area cooperative caching

Shark's main claim is that it improves a file server's scalability, while retaining its benefits. We now study the end-to-end performance of reads in a cooperative environment with many clients attempting to simultaneously read the same file(s).

In this section, we evaluate Shark on Emulab [36]. These experiments allowed us to evaluate various coop-

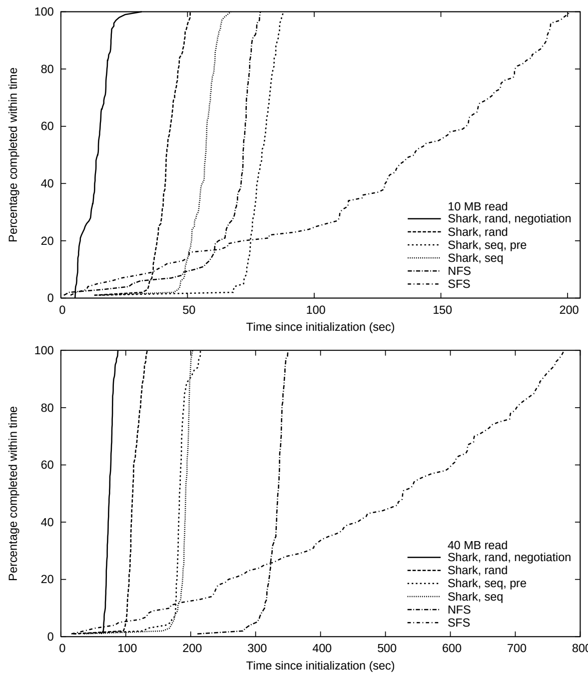

Figure 8: *Client latency.* Time (seconds) for ∼100 LAN hosts to read a 10 MB (top) and 40 MB (bottom) file.

erative strategies in a better controlled environment. In all the configurations of Shark, clients attempt to download a file from four other proxies simultaneously.

Figure 8 shows the cumulative distribution functions (CDFs) of the time needed to read a 10 MB and 40 MB (random) file across 100 physical Emulab hosts, comparing various cooperative read strategies of Shark, against vanilla SFS and NFS. In each experiment, all hosts mounted the server and began fetching the file simultaneously. We see that Shark achieves a median completion time $< \frac{1}{4}$ that of NFS and $< \frac{1}{6}$ that of SFS. Furthermore, its 95th percentile is almost an order of magnitude better than SFS.

Shark's fast, almost vertical rise (for nearly all strategies) demonstrates its cooperative cut-through routing: Shark clients effectively organize themselves into a distribution mesh. Considering a single data segment, a client is part of a chain of nodes performing cut-through routing, rooted at the origin server. Because clients may act as root nodes for some blocks and act as leaves for others, most finish at almost synchronized times. The lack of any degradation of performance in the upper percentiles demonstrates the lack of any heterogeneity, in terms of both network bandwidth and underlying disk/CPU load, among the Emulab hosts.

Interestingly, we see that most NFS clients finish at loosely synchronized times, while the CDF of SFS clients' times has a much more gradual slope, even though

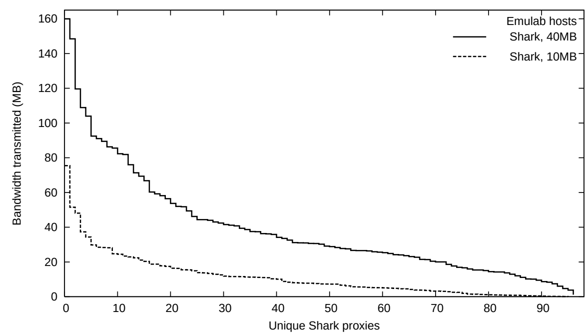

Figure 9: *Proxy bandwidth usage.* MBs served by each Emulab proxy when reading 40 MB and 10 MB files.

both systems send all read requests to the file server. Subsequent analysis of NFS over TCP (instead of NFS over UDP as shown) showed a similar slope as SFS, as did Shark without its cooperative cache. One possible explanation is that the heavy load on (and hence congestion at) the file server imposed by these non-cooperative file systems drives some TCP connections into back-off, greatly reducing fairness.

We find that a $random$ request strategy, coupled with inter-proxy $negotiation$, distinctly outperforms all other evaluated strategies. A sequential strategy effectively saw the clients furthest along in reading a file fetch the leading (four) chunks from the origin file server; other clients used these leading clients as proxies. Thus, modulo possible inter-proxy timeouts and synchronous requests in the non-pre-announce example, the origin server saw at most four simultaneous chunk requests. Using a $random$ strategy, more chunks are fetched from the server simultaneously and thus propagate more quickly through the clients' dissemination mesh.

Figure 9 shows the total amount of bandwidth served by each proxy as part of Shark's cooperative caching, when using a random fetch strategy with inter-proxy negotiation for the 40 MB and 10 MB experiments. We see that the proxy serving the most bandwidth contributed four and seven times more upstream bandwidth than downstream bandwidth, respectively. During these experiments, the Shark file server served a total of 92.55 MB and 15.48 MB, respectively. Thus, we conclude that Shark is able to significantly reduce a file server's bandwidth utilization, even when distributing files to large numbers of clients. Furthermore, Shark ensures that any one cooperative-caching client does not assume excessive bandwidth costs.

4.4 Wide-area cooperative caching

Shark's main claim is that it improves a file server's scalability, which still maintaining security, accountability, etc. In our cooperative caching experiment, we study the end-

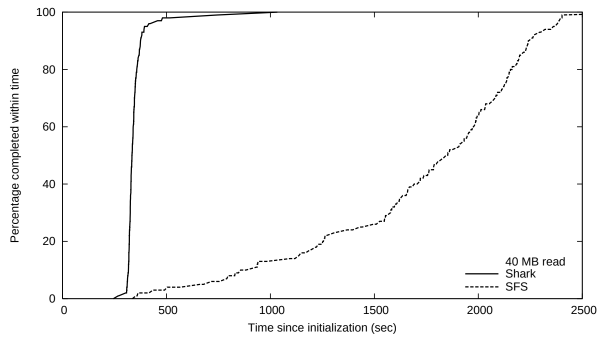

Figure 10: *Client latency.* Time (seconds) for 185 hosts to finish reading a 40 MB file using Shark and SFS.

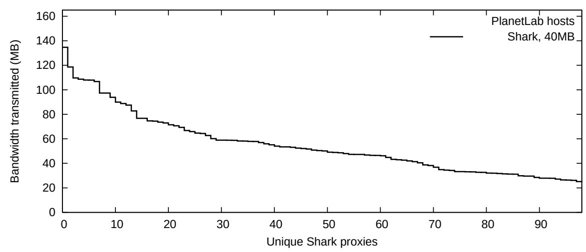

Figure 11: *Proxy bandwidth usage.* MBs served by each PlanetLab proxy when reading 40 MB files.

to-end performance of attempting to perform reads within a large, wide-area distributed test-bed.

On approximately 185 PlanetLab hosts, well-distributed from North America, Europe, and Asia, we attempted to simultaneously read a 40 MB random file. All hosts mounted the server and began fetching the file simultaneously.

Figure 10 shows a CDF of the time needed to read the file on all hosts, comparing Shark with SFS.

% done in (sec)	50%	75%	90%	95%	98%
Shark	334	350	375	394	481
SFS	1848	2129	2241	2364	2396

We see that, between the 50th and 98th percentiles, Shark is five to six times faster than SFS. The graph's sharp rise and distinct knee demonstrates Shark's cooperative caching: 96% of the nodes effectively finish at nearly the same time. Clients in SFS, on the other hand, complete at a much slower rate.

Wide-area experiments with NFS repeatedly crashed our file server (*i.e.*, it caused a kernel panic). We were therefore unable to evaluate NFS in the wide area.

Figure 11 shows the total amount of bandwidth served by each proxy during this experiment. We see that the proxy serving the most bandwidth contributed roughly three times more upstream than downstream bandwidth.

Figure 12 shows the number of bytes read from our file server during the execution of these two experiments. We

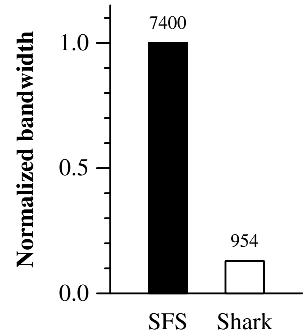

Figure 12: *Server bandwidth usage.* Megabytes read from server as a 40 MB file is fetched by 185 hosts.

see that Shark reduces the server's bandwidth usage by an order of magnitude. In fact, we believe that Shark's client cache implementation can be improved to reduce bandwidth usage quite further: We are currently examining the trade-offs between continually retrying the cooperative cache and increased client latency.

5 Related Work

There are numerous network file systems designed for local-area access. NFS [31] provides a server-based file system, while AFS [14] improves its performance via client-side caching. Some network file systems provide security to operate on untrusted networks, including AFS with Kerberos [32], Echo [18], Truffles [27], and SFS [21]. Even wide-area file systems such as AFS do not perform any bandwidth optimizations necessary for types of workloads and applications Shark targets. Additionally, although not an intrinsic limitation of AFS, there are some network environments that do not work as well with its UDP-based transport compared to a TCP-based one. This section describes some complementary and alternate designs for building scalable file systems.

Scalable file servers. JetFile [12] is a wide-area network file system designed to scale to large numbers of clients, by using the Scalable Reliable Multicast (SRM) protocol, which is logically layered on IP multicast. JetFile allocates a multicast address for each file. Read requests are multicast to this address; any client which has the data responds to such requests. In JetFile, any client can become the manager for a file by writing to it—which implies the necessity for conflict-resolution mechanisms to periodically synchronize to a storage server—whereas all writes in Shark are synchronized at a central server. However, this practice implies that JetFile is intended for read-write workloads, while Shark is designed for read-heavy workloads.

High-availability file systems. Several local-area systems propose distributing functionality over multiple collocated hosts to achieve greater fault-tolerance and availability. Zebra [13] uses a single meta-data server to serialize meta-data operations (*e.g.* i-node operations), and maintains a per-client log of file contents striped across multiple network nodes. Harp [17] replicates file servers to ensure high availability; one such server acts as a primary replica in order to serialize updates. These techniques are largely orthogonal to, yet possibly could be combined with, Shark's cooperative caching design.

Serverless file systems. Serverless file systems are designed to offer greater local-area scalability by replicating functionality across multiple hosts. xFS [3] distributes data and meta-data across all participating hosts, where every piece of meta-data is assigned a host at which to serialize updates for that meta-data. Frangipani [34] decentralizes file-storage among a set virtualized disks, and it maintains traditional file system structures, with small meta-data logs to improve recoverability. A Shark server can similarly use any type of log-based or journaled file system to enable recoverability, while it is explicitly designed for wide-area scalability.

Farsite [1] seeks to build an enterprise-scale distributed file system. A single primary replica manages file writes, and the system protects directory meta-data through a Byzantine-fault-tolerant protocol [7]. When enabling cross-file-system sharing, Shark's encryption technique is similar to Farsite's convergent encryption, in which files with identical content result in identical ciphertexts.

Peer-to-peer file systems. A number of peer-to-peer file systems—including PAST [30], CFS [8], Ivy [23], and OceanStore [16]—have been proposed for wide-area operation and similarly use some type of distributed-hash-table infrastructure ([29, 33, 37], respectively). All of these systems differ from Shark in that they provide a serverless design: While such a decentralized design removes any central point of failure, it adds complexity, performance overhead, and management difficulties.

PAST and CFS are both designed for read-only data, where data (whole files in PAST and file blocks in CFS) are *stored* in the peer-to-peer DHT [29, 33] at nodes closest to the key that names the respective block/file. Data replication helps improve performance and ensures that a single node is not overloaded. In contract, Shark uses Coral to *index* clients caching a replica, so data is only cached where it is needed by applications and on nodes who have proper access permissions to the data.

Ivy builds on CFS to yield a read-write file system through logs and version vectors. The head of a per-client log is stored in the DHT at its closest node. To enable multiple writers, Ivy uses version vectors to order records from different logs. It does not guarantee read/write consistency. Also managing read/write storage via versioned logs, OceanStore divides the system into a large set of untrusted clients and a core group of trusted servers, where updates are applied atomically. Its Pond prototype [28] uses a combination of Byzantine-fault-tolerant protocols, proactive threshold signatures, erasure-encoded and block replication, and multicast dissemination.

Large file distribution. BitTorrent [5] is a widely-deployed file-distribution system. It uses a central server to track which clients are caching which blocks; using information from this meta-data server, clients download file blocks from other clients in parallel. Clients access BitTorrent through a web interface or special software.

Compared to BitTorrent, Shark provides a file-system interface supporting read/write operations with flexible access control policies, while BitTorrent lacks authorization mechanisms and supports read-only data. While BitTorrent centralizes client meta-data information, Shark-stores such information in a global distributed index, enabling cross-file-system sharing (for world-readable files) and taking advantage of network locality.

6 Conclusion

We argue for the utility of a network file system that can scale to hundreds of clients, while simultaneously providing a drop-in replacement for local-area file systems. We present Shark, a file system that exports existing local file systems, ensures compatibility with existing administrative procedures, and provides performance competitive with other secure network file systems on local-area networks. For improved wide-area performance, Shark clients construct a locality-optimized cooperative cache by forming self-organizing clusters of well-connected machines. They efficiently locate nearby copies of data using a distributed index and stripe downloads from multiple proxies. This simultaneously reduces the load on file servers and delivers significant performance improvements for the clients. In doing so, Shark appears promising for achieving the goal of a scalable, efficient, secure, and easily-administered distributed file system.

Acknowledgments. We thank Vijay Karamcheti, Jinyuan Li, Antonio Nicolosi, Robert Grimm, our shepherd, Peter Druschel, and members of NYU systems group for their helpful feedback on drafts of this paper. We would like to thank Emulab (Robert Ricci, Timothy Stack, Leigh Stoller, and Jay Lepreau) and PlanetLab (Steve Muir and Larry Peterson) researchers for assistance in running file-system experiments on their test-beds, as well as Eric Freudenthal and Jayanth Kumar Kannan for remote machine access. Finally, thanks to Jane-Ellen Long at USENIX for her consideration.

This research was conducted as part of the IRIS project (`http://project-iris.net/`), supported by the NSF under Cooperative Agreement No. ANI-0225660. Michael Freedman is supported by an NDSEG Fellowship. David Mazières is supported by an Alfred P. Sloan Research Fellowship.

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 *OSDI*, Boston, MA, December 2002.

[2] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient overlay networks. In *SOSP*, pages 131–145, Banff, Canada, October 2001.

[3] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roseli, and R. Y. Wang. Serverless network file systems. *ACM Trans. on Computer Systems*, 14(1):41–79, February 1996.

[4] M. Bellare, R. Canetti, and H. Krawczyk. Keyed hash functions and message authentication. In *Advances in Cryptology—CRYPTO '96*, Santa Barbara, CA, August 1996.

[5] BitTorrent. http://www.bittorrent.com/, 2005.

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

[7] M. Castro and B. Liskov. Proactive recovery in a byzantine-fault-tolerant system. In *OSDI*, San Diego, October 2000.

[8] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *SOSP*, Banff, Canada, October 2001.

[9] D. H. J Epema, Miron Livny, R. van Dantzig, X. Evers, and Jim Pruyne. A worldwide flock of condors: Load sharing among workstation clusters. *J. Future Generations of Computer Systems*, 12:53–65, 1996.

[10] M. J. Freedman, E. Freudenthal, and D. Maziéres. Democratizing content publication with Coral. In *NSDI*, San Francisco, CA, March 2004.

[11] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP*, Litchfield Park, AZ, December 1989.

[12] B. Grönvall, A. Westerlund, and S. Pink. The design of a multicast-based distributed file system. In *OSDI*, New Orleans, LA, February 1999.

[13] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. In *SOSP*, Asheville, NC, December 1993.

[14] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Trans. on Computer Systems*, 6(1):51–81, February 1988.

[15] H. Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is ssl?). In *Advances in Cryptology—CRYPTO 2001*, Santa Barbara, CA, August 2001.

[16] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *ASPLOS*, Cambridge, MA, November 2000.

[17] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. eplication in the Harp file system. *Operating Systems Review*, 25(5):226–238, October 1991.

[18] T. Mann, A. D. Birrell, A. Hisgen, C. Jerian, and G. Swart. A coherent distributed file cache with directory write-behind. *ACM Trans. on Computer Systems*, 12(2):123–164, May 1994.

[19] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS*, Cambridge, MA, March 2002.

[20] D. Mazières. A toolkit for user-level file systems. In *USENIX*, Boston, MA, June 2001.

[21] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *SOSP*, Kiawah Island, SC, December 1999.

[22] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *SOSP*, October 2001.

[23] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *OSDI*, Boston, MA, December 2002.

[24] PlanetLab. http://www.planet-lab.org/, 2005.

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

[26] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *ACM SIGCOMM*, San Diego, CA, August 2001.

[27] P. Reiher, Jr. T. Page, G. J. Popek, J. Cook, and S. Crocker. Truffles — a secure service for widespread file sharing. In *PSRG Workshop on Network and Distributed System Security*, San Diego, CA, February 1993.

[28] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore prototype. In *FAST*, Berkeley, CA, March 2003.

[29] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware*, November 2001.

[30] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *SOSP*, Banff, Canada, October 2001.

[31] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *Summer 1985 USENIX*, Portland, OR, June 1985.

[32] J. G. Steiner, B. C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Winter 1988 USENIX*, Dallas, TX, February 1988.

[33] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. In *IEEE/ACM Trans. on Networking*, 2002.

[34] C. Thekkath, T. Mann, and E Lee. Frangipani: A scalable distributed file system. In *SOSP*, Saint Malo, France, October 1997.

[35] A. Westerlund and J. Danielsson. Arla—a free AFS client. In *1998 USENIX, Freenix track*, New Orleans, LA, June 1998.

[36] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI*, Boston, MA, December 2002.

[37] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE J. Selected Areas in Communications*, 22(1):41–53, 2003.