Starfish: Resilient Distributed Data Structures in Untrusted Environments

John Bicket, Sanjit Biswas and Anjali Gupta
{jbicket, biswas, anjali@mit.edu

Abstract

This paper presents Starfish, a system for maintaining data structures in distributed environments without central authorities. Using the abstraction of a hash table, clients construct application-specific data structures without worrying about the underlying service properties necessary for operation in untrusted and dynamic environments. Starfish tolerates faulty servers by replicating data and using a two-phase Byzantine quorum protocol for access. Data is periodically regenerated to recover from replica failure and is cryptographically signed to detect tampering. Users establish trust with servers using certificate chains verified by client provided functions.

Our design exposes performance trade-offs. clients determine the degree of replication, form of protection and other implementation details appropriate for their setting. We evaluate the performance of a distributed file system built using Starfish and discuss other potential applications. Preliminary results indicate our file system's performance is competitive with unreplicated NFS.

1 Introduction

Traditionally, applications have relied on servers to provide a common name space, protection and reliable storage for shared data. Due to their importance, these servers are generally well maintained and built from redundant components but are still prone to becoming single points of failure. Centralized servers rely on a dedicated administrative staff to maintain reliability and are vulnerable to geographically localized faults such as power outages or network failures. Furthermore, their store of sensitive and valuable information makes them attractive targets for attack, thus demanding a high level of administrator competence and attention.

In designing Starfish, our goal has been to utilize the collective resources of unreliable machines such as user desktop computers to provide scalable, available and consistent storage with a general interface. Earlier systems such as DDS [4] and Frangipani [11] had similar intentions, but were not widely adopted due to dependence on a centralized lookup service and a trusted setting such as a cluster within a secured machine room.

Advances in peer-to-peer lookup mechanisms such as Chord [10] and Pastry [9] have eliminated the need for a central directory of servers but their existing storage layer lacks the necessary protection for operation in untrusted environments. Recent systems built on peer-to-peer lookup mechanisms (Ivy, Farsite) have demonstrated that operation in such environments without central authorities is feasible, but they both provide the abstraction of a distributed file system. Such an interface is not sufficiently general for all applications, which may desire an alternative data structure or forms of protection. Examples include a shared calendaring application, a B-tree based database or music sharing system which are not naturally adapted to a file system abstraction.

In this paper, we present a self-managing storage layer designed to maintain distributed data structures in untrusted and dynamic environments. Unlike file systems interfaces which limit application flexibility, Starfish presents the abstraction of a distributed hash table with the two primitives `write(blockid, data, credential)` and `read(blockid)` from which authors can design data structures specific to their application. Behind the abstraction barrier, Starfish implements the complex mechanisms used to access, replicate and recover data.

Starfish maintains availability of data through replication. Each block is stored on a group of servers, determined by the block id, which are accessed using a two-phase Byzantine quorum protocol. The protocol tolerates up to b failures given $3b + 1$ replicas, which may be the result of server failure or less commonly malicious tampering of data. Starfish recovers from server failure by periodically regenerating missing blocks. Since servers are untrusted, clients encrypt data to ensure privacy and verify signatures to detect tampering.

Starfish protects blocks from unauthorized modification with client provided functions which verify user certificate chains. Using cryptographic techniques, servers can verify a user's authenticity without the presence of online authorities. Client provided functions allow servers to protect blocks without understanding the contents. These functions also implement guarded writes, which enable users to build support for high-level operations such as transactions. We demonstrate the ability to implement complex data structures by presenting StarFS, a shared distributed file system built using Starfish.

We have implemented all of the design presented in this paper with the exception of replication. Although we have support for verifying credentials, time constrains prevented us from implementing security in StarFS.

2 Distributed Hash Tables

Distributed Hash Tables (DHTs)[10][9] have been a central area of research interest because of their fault-tolerant and decentralized properties. DHTs provide a distributed object location service which enables clients to identify which computer is responsible for a given key. Primarily, they have two operations:

- $get(key) \rightarrow value$
- $put(key, value)$

Existing file systems such as CFS [3] rely on a single writer to publish the entire file system as blocks where the key of a particular block was the content hash of the data stored there. This allows clients to verify the integrity of data they retrieve from the hash table and allow blocks to be cached since they are inherently immutable. Unfortunately, such schemes are limited to static distribution systems; if modifications were allowed, the key of the altered block would change, forcing any blocks that pointed to the key to change as well. This continues until the change to propagates up to the root of the file system, resulting in frequent changes to the root (and hence it's key as well).

Other systems such as Ivy [8] use public key blocks, where the key of the block is based on the public key an asymmetric key pair. In this case, the writer signs the blocks using their private key, which enables clients to verify the integrity of the block. Public key blocks have two major drawbacks: malicious servers can present clients with stale copies of blocks and each public key block requires a corresponding private key. As a result, it is difficult to build general data structures solely from these blocks.

3 Replication in Starfish

When dealing in a distributed system with mutable data, a single element in the system cannot be entrusted with information as this presents a single point of failure. Systems with mutable data need mechanisms for verifying the existence of data and ensuring integrity without depending on any particular node. To ensure the validity of data, Starfish uses replication to mask out faulty nodes, allowing data to survive even in a few of the nodes it is stored on behave incorrectly.

The advent of Byzantine agreement [1] and its recent applications to redundant file storage have used commit mechanisms that tolerate b Byzantine failures out of $3b + 1$ replicas. File systems built upon these agreement schemes usually only require one round trip messages between a client performing a write and a primary replica in the replica group, but require $O(n^2)$ messages to be passed between the replicas to resolve a commit.

Byzantine quorum systems[6], which do not implement a replicated state machine, take a less heavy-handed approach to commits; the client initiates a standard two phase commit protocol with the replicas. The client sends each replica a prepare message, and if enough respond with successes, it proceeds to send all the servers commit message. Since Byzantine quorum does not use a primary replica to order the client requests, reservation mechanisms are necessary to ensure mutual exclusion and strict ordering for client commits. Chockler, et. al present a protocol with these features [2] that tolerates b malicious servers and b network failures (which result in stale servers).

These quorum protocols have desirable communications overhead since replicas do not have to communicate with each other and commits take only $O(n)$ messages. But these quorum protocols do not tolerate client failure; if a client fails half way through the commit phase, inconsistent data may appear to readers until the next write. The rest of this section will detail the quorum protocol for Starfish blocks and how they ensure consistent data during client failures.

3.1 Identifying Replicas

Systems that use a form of Byzantine agreement or Byzantine quorum must have a method for providing consistent identification of members of a quorum group. This is essential to ensuring that when a client performs a read he gets data from the same replica servers that performed a write; otherwise, faulty servers can "pass on" their faults to otherwise non-faulty nodes.

Other systems such as OceanStore[5] resort to storing all mutable meta-data on a central set of servers that remains static over time.

Instead of relying on any trusted components, Starfish relies on the underlying distributed hash table protocol to reliably identify who is responsible for particular keys. Recent work on hardening DHTs against Byzantine faulty servers [7] has led us to believe such services are feasible. It assumes that a client file system knows the relative join/leave rates of the overlay network and can adjust the number of faulty nodes b it can tolerate based on both the join/leave rate and the proportion of faulty nodes in the system.

Each Starfish block has $r = 3b + 1$ replicas, where b is the number of faulty nodes our modified Byzantine quorum protocol can tolerate. b should be chosen based on the estimated join/leave rate of the servers in the distributed hash table as well as the proportion of Byzantine faulty nodes in a system.

We assume that for each block r can be found by the client in parent meta-data or, in the case of the root meta-data block, out of band. For a particular block that resides at key, Starfish keeps replicas at

$$\text{SHA1}(key), \text{SHA1}(key + 1) \ldots \text{SHA1}(key + r - 1)$$

3.2 Writes and Distributed Mutual Exclusion

Starfish uses a modified Byzantine quorum protocol [6] to ensure consistency of mutable blocks in distributed hash tables.

The Starfish write contains two phases: reserve and commit. For each block, every replica server tracks the last reservation for a Starfish block. Upon receiving a reservation message, the replica server checks the if another client has reserved the block; if there is no outstanding reservation the replica server returns success, and otherwise it returns failure.

The write protocol operates on the client side as follows:

1. Send reserve message to all $3b + 1$ replicas
2. Wait for $2b + 1$ successful responses
3. Send commit message to all $3b + 1$ replicas
4. Wait for at least $2b + 1$ successful responses

If the client does not receive at least $2b + 1$ successful responses to the reserve message, it backs off for a random interval. Requiring at least $2b + 1$ successful reservations before proceeding ensures that no more than one cooperating client will proceed to the commit phase.

In the worst case with contending clients, the b faulty servers will always return success for a reservation message; of the $2b + 1$ servers left, one node will receive at least $b + 1$ of them, allowing it to proceed since it received $2b + 1$ successful responses. All other clients will not be able to assemble more than $2b$ successful responses, and will be forced to back off, ensuring that no more than two clients will be able to enter the commit phase at the same time.

Obviously the choice for the reservation time must be a very conservative estimate of the duration for how long the commit phase should take. Choosing too large a reservation time only affects write performance for highly contended blocks.

On the other hand, if b faulty servers always return failure for a reservation message, $2b+1$ servers are left that are operating properly. This means that a client must receive successful reservations from all of the non-faulty servers, but it does ensure that up to be b

nodes may be faulty without preventing a node from committing writes.

As described in later sections, Starfish has has provisions for guarded writes, which when combined with the mutual exclusion protocol can be used to construct locks for higher

3.3 Client Reads and Failure Detection

Replica servers allow anyone to read Starfish blocks; we assume that any file system that needs read protection for their data will use cryptography. The read protocol operates form the client side as follows:

1. Send read request to all $3b + 1$ replicas

2. Wait for $2b + 1$ responses.

Normal Byzantine quorum protocols are vulnerable to client failures; if a client crashes before writing to $2b + 1$ replica servers, clients that read at a later time by waiting for $2b + 1$ agreeing replicas may retrieve different data depending on which replica servers they receive messages from first. This inconsistency must never be allowed to occur; when a client write crashes mid-commit, either 1) the rest of the write should be committed or 2) all the servers should revert to the old block.

To ensure these guarantees Starfish uses the monotonically increasing version numbers allows clients to differentiate between the new data that was partially written to the replicas and the previous data. Armed with this, clients executing a read can detect a partial write occurred if $2b + 1$ replicas don't agree on the most recent commit. Using the `version()` function, the client can deduce the failed update and then perform a write of that update the stale servers since the credentials are all contained within the block.

Note that the credentials are block specific; each version number is signed by the writer, ensuring that it is computationally unfeasable for a malicious client to produce a valid block unless he has the private key that is verified by the certificate chain.

Using version numbers provides Starfish blocks with a write-forward-only mechanism; stale replica servers can be brought up to speed with a single update.

3.4 Regeneration

When a replica server becomes responsible for a new part of the key space, either by joining a DHT or by other nodes leaving, it must determine which Starfish key it is responsible for and then must determine what blocks it should store under those keys. The new replica server needs to avoid inheriting any faults from the node that was previously responsible for the key; to determine what the block value is he simply does a read. To ensure that new replica servers will be informed of their newfound responsibility, each should periodically perform a read. This read serves two functions; first, it alerts new nodes that they should be storing a particular key. Second, it ensures that a node does not miss an update and allows replica servers to recover from faulty behavior. Note that if enough nodes leave the DHT a read may not be possible; this is why the amount of faults tolerated, b, must take into account the join/leave rate.

3.5 Partition

Dealing with network partitions is difficult for any distributed system dealing with mutable data; it is hard enough even distinguish network partition from massive node failure. In order to achieve consistency, Starfish reads and writes will not succeed if less than $2b + 1$ of the replicas are on the same side of the partition. Requiring over $\frac{2}{3}$ of the replicas to be present for operation ensures that at the time the partition occurred no more than one partition will be able to make progress. As time moved on, the only way that both sides would be able to proceed is if all the nodes on the majority side regenerated and then $\frac{1}{3}$ of them moved to the minority side. We assert that such a case would be very rare.

4 Client Provided Functions

Starfish takes advantage of two client-provided functions to reason about meta-data, `audit()` and `version()`. These functions allow replica servers to authenticate writes without knowing the underlying format of the data blocks. These functions run on untrusted java byte-code and are sandboxed by a virtual machine. They are allowed to only use provided functions (a small set of cryptographic verification algo-

Field	Use
Key	Block Identification
Root public key	Verifying certificates
Permissions block key	Decrypting block data
Certificate of last writer	Integrity check
Audit function byte-code	Authentication
Version number (signed)	Freshness information
Block data (encrypted)	

Figure 1: Starfish block used for StarFS meta-data

Public Key	Encrypted Symmetric Key
pub_1	$(group - key)_{pub_1}$
pub_2	$(group - key)_{pub_2}$
pub_3	$(group - key)_{pub_3}$

Figure 2: Permissions block

rithms). They are allowed no network access and have limits on how much CPU time they can use.

`audit()` is called by the server when presented with a write request. This function returns whether the new block is a valid replacement for the old block. Audit functions are expected to check cryptographic certificate chains and verify the root certificate of the new block remains the same in the new block when determining whether a write request is valid. Replica servers call `audit()` to determine whether a write request should be granted for both reservations and commits. `version(oldblock, newblock)` is used by clients when performing reads from various replicas to determine the latest version of a particular block. Version must must be presented with a previous block to ensure that the root certificates that authenticate the credential chains do not change over time.

Through the use of the client provided functions, only clients that possess the correct credentials can make any progress with the replica servers. We assume that if a client has the correct credentials, it is in the client's best interest to follow the protocol correctly (i.e. commit only after it has successfully completed the reservation stage). Starfish does not safeguard against malicious authorized clients, as they could simply remove the block or write garbage to the block as easily as it could make the data inconsistent.

5 Starfish File System

As a demonstration of a complex data structure built on Starfish, we built StarFS, a distributed file system with NFS-like semantics.

The implementation acts as a local loopback NFS v3 server, in cooperation with the host's in-kernel NFS client support. A block key is also used as its NFS file handle.

StarFS frees the storage system from the responsibility of checking the integrity of data. As described earlier, it also provides the storage layer with the byte-code for an `audit()` function by which to authenticate users of the system. The storage layer simply performs authentication by executing this function with a certificate or *credential chain* provided by the user. This allows the structure of the file system to be completely transparent to the storage layer, allowing complete flexibility in the design of data structures.

StarFS consists of a rooted tree of linked blocks which store all the data and metadata of the file system. The block size can be variable, and is a client option. The client can also explicitly state the degree of replication of each block. Thus, more popular or more important blocks can be replicated to a large extent without incurring the same cost on other blocks in the system. For example, the root block of the file system may be heavily replicated, but less important file or directory i-nodes may have fewer replicas to increase performance.

5.1 Block Structures

StarFS consists of two kinds of blocks - meta-data blocks and data blocks. All blocks are identified using 160-bit keys. meta-data blocks are used to store file i-nodes and other meta-data structures such as userid's and permissions tables. They are named by randomly generated keys, which balances their storage load over a random set of storage servers. The blocks are cryptographically signed to ensure integrity. Table 1 shows the various fields in a meta-data block and their use.

Data blocks just store data and are named by collision-resistant cryptographic hashes of their contents. The self-certifying nature of these blocks allows clients to proceed without performing a quorum read.

StarFS has file-level protection, it records file owners (including the notion of group ownership) and permission modes. Every user has a public-private key pair and the public key uniquely identifies her in the system. Permissions are enforced only for block modification and extension. A user who wants to keep her data unreadable should encrypt the data. StarFS provides an infrastructure for sharing the encryption key securely with a group of users.

5.2 Using StarFS

In this section, we describe how StarFS performs some selected operations.

5.2.1 File system creation.

StarFS builds a new file system by creating a root block and a set of permissions blocks for the file system. A permissions block corresponds to a group of users having read permissions on the same set of blocks. Read permissions are enforced using symmetric key encryption, and permissions blocks help in sharing this key securely with a group of users. For each user in the group, the block contains the encryption key which is encrypted, in turn, with the user's public key. Figure 2 shows the structure of a permissions block.

The root block corresponds to the i-node of the root directory that is exported to the world and it is keyed by a randomly generated number. The user then mounts the local StarFS server as an NFS file system using the the root block key as the NFS root file handle.

5.2.2 File creation.

When a user creates a new file, the kernel NFS client code sends the local StarFS server an NFS CREATE request. The request contains the directory i-node key and a file name. StarFS reads the directory i-node block, and its indirect blocks (if required) and appends an entry (a randomly generated key) for the new file. StarFS also creates and initializes a new meta-data block and writes it to storage under that key. StarFS returns the new file's i-node key in a file handle to the NFS client.

5.2.3 File Read.

An NFS READ request contains the file's i-node key, an offset within the file and the number of bytes to read. StarFS first reads the i-node block and then the data blocks as required.

Reads in StarFS proceed very differently for meta-data blocks and data blocks. Meta-data blocks, such as file i-nodes contain all information for checking freshness, authentication and integrity of data. They also contain pointers to the latest data blocks. Thus, no freshness checks are needed while reading data blocks. Also, data blocks are keyed by the hash of their content and are thus self-certifying. This makes reading data blocks much cheaper than meta-data blocks. Since data blocks are expected to be much larger in number than meta-data blocks for a typical read sequence, the cost gets amortized over a large number of blocks, making an overall file read cheap.

A meta-data block is read by a Starfish retrieve operation. As described in a previous section, this involves reading 2b+1 out of 3b+1 replicas and checking their authenticity. Authenticity is checked by cryptographically verifying the block with the last writers public key which is included in the unsigned part of the block. StarFS then chooses the most recent copy by comparing the signed version numbers of the 2b+1 blocks. After obtaining the most recent block, StarFS retrieves the key of the permissions block corresponding to that file or directory (refer Figure 1) to find the symmetric key which has been used to encrypt the data portion in the block. StarFS reads this block and returns it to the user. The user (or typically a trusted agent process) reads the entry corresponding to her public key pub, decrypts the symmetric encryption key sym using her private key $priv$ and returns sym to StarFS. StarFS then uses sym to decrypt the data portion in the block.

The data portion of the file i-node contains pointers to direct and indirect blocks. StarFS computes the blocks to be read using the file offset specified in the NFS request and issues a retrieve to the storage layer for these blocks.

The data blocks are read by an ordinary retrieve operation. This involves reading blocks from replicas until one block is received whose key matches the hash of its contents.

5.2.4 File Write

An NFS WRITE request contains the file's i-node key, an offset within the file where the write should start, the number of bytes to be written and the data to be written. StarFS has a block replacement policy for writes, and therefore it reads the i-node block and relevant indirect and data blocks affected by the write before making the change and writing the blocks back to storage. It also increments the version number after every change. Not only are version number monotonically increasing, the increment is also fixed. This puts the version numbers in one-to-one correspondence with their order of existence in the system, and helps in detection of version number collisions.

Writes are authenticated by the storage layer using a function provided by StarFS or by the client. This function called `audit()` is part of every metadata block and takes four arguments supplied by the StarFS when it sends a write request:

- *Block key*
- *Certificate* or credential chain showing that the user is authorized to make the write. Its structure and use is described further ahead in this section.
- *New Block* or the data that needs to be written out.
- $(hash(newblock), version)_{priv-key}$ ie a copy of the new block and the updated version number signed by the users private key.

Guarded writes. Similar to the case of file reads, writes are more expensive and more likely to face contention for meta-data blocks. Meta-data block keys remain unchanged after the write, and the block is stored with the same servers as before. Any previous writes are written over. Thus every writer writing a meta-data block has to check that it is not writing over some write that has happened since it last read the block. This is done by checking that the new version number is strictly greater than the old one. This is sufficient because of the previous argument of fixed increments in version numbers.

A similar problem does not exist for data blocks as they are keyed by their content. Every time the content changes, the identity of the block gets changed and thus no old writes are destroyed. The additional benefit of content-hash blocks is they eliminate the storage of redundant data blocks.

Authentication. This is done using certificates. There is flexibility in our system to use certificate chains which allow a hierarchical distribution of permissions. Let us consider a one-level chain: The file system is created by a user X with public key pub_x and private key $priv_x$. User X gives write permissions for all files to User Y with public key pub_y and private key $priv_y$.

X generates a permission P_y = "$pub_y + write$". X gives Y the certificate $C_y = (P_y, pub_y)_{priv_x}$ ie the certificate contains the permission and the public key of Y signed by X. The public key of X is stored in every meta-data block, so the `audit` function can extract the public key of Y from this certificate. `audit` then verifies the signed copy of the hash and the version number provided in the request. Furthermore, the new block is hashed and this hash is checked against the signed hash. If it matches, the write is allowed to proceed.

This one-level chain can be extended over many levels, each representing a signed capability, finally signed by the "root" private key.

Replay Attacks. As we saw for authentication, all verification information is sent over the wire unencrypted since StarFS does not trust the storage servers. We also do not make any synchrony assumptions for the system. This requires our system to be robust against replay attacks. The signed version number sent over as part of the request protects the system against such attacks since writes need to have a strictly greater version number to be accepted. The replay will fail because the version number in storage will be at least as high as the one in the replayed request, or would be a replay of a failed write which would fail again.

6 Implementation and Performance

Starfish blocks and the two-phase commit protocol have been implemented in DHash [3] which runs on top of the Chord [10] routing protocol.

Using the SFS user level file system toolkit, we have built a working file system with NFS-like semantics. Due to an implementation artifact involving parallel updates to meta-data using the NFS client, reservations were turned off to obtain performance numbers. For the NFS unreplicated server results, we connected the NFS server running Red Hat Linux 7.3 to another workstation running FreeBSD 4.5 over switched 100-Mbit Ethernet. To obtain the measurements for StarFS, we used FreeBSD 4.5 running our modified version of Chord and Dhash. We ran the read and write phase of the Modified Andrew Benchmark to obtain performance numbers shown in **Figure 3**.

Most of the files in the benchmark use fewer than three blocks. As a result, meta-data blocks are an artificially large portion of the total file size. This impedes general performance as we contact several replicas for each operation. The devastatingly poor write performance is due to meta-data updates for each block written in the file. If we had implemented open-to-close consistency, these results would be significantly better and comparable to the penalties incurred on reads.

7 Future Work

Information such as join/leave rates, independence of failure, and other measurements about peer to peer systems need to be obtained to further explore the feasibility of peer to peer storage. More performance measurements need to be taken and building a reliable Starfish system would be a particularly interesting undertaking. We have started thinking about data recovery in the case of massive node failures and dealing more effective with partitions. Also, quotas and garbage collection would be needed for a system to support a large number of users for a long period of time.

8 Conclusion

Starfish provides a scalable, available and consistent storage layer built on a collection of unreliable and insecure machines. Starfish provides the features of a common name space, protection and reliability associated with centralized servers, without their susceptability to geographically centralized faults and administrative overhead.

Starfish replicates data on randomly chosen groups of servers and uses a two-phase Byzantine quorum protocol for access. Replica servers authenticate users' credentials using a client provided `audit()` function before committing writes. Clients ensure data secrecy using encryption.

By using a distributed hash table interface where clients perform block replacement, Starfish provides service authors with the flexibility to implement a wide array of application specific data structures. Clients are free to choose the degree of replication, consistency semantics, data structures specifics and form of protection appropriate for their setting.

In our case study of StarFS, we have shown it possible build complex data structures on Starfish. Our preliminary results show the performance of our file system is competitive with unreplicated NFS.

References

[1] Castro and Liskov. Practical byzantine fault tolerance. In *OSDI: Symposium on Operating Systems Design and Implementation*. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS, 1999.

[2] G. Chockler, D. Malkhi, and M. Reiter. Backoff protocols for distributed mutual exclusion and ordering. pages 11–20.

[3] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[4] S. Gribble, E. Brewer, J. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction, 2000.

[5] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[6] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. pages 569–578, 1997.

[7] Ayalvadi Ganesh Antony Rowstron Miguel Castro, Peter Druschel and Dan S. Wallach. Security

for structured peer-to-peer overlay networks. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[8] Athicha Muthitacharoen, Robert Morris, Thomer Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[9] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–??, 2001.

[10] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[11] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *Symposium on Operating Systems Principles*, pages 224–237, 1997.

	Unreplicated NFS	**StarFS 1 replica**	**StarFS 4 replicas**	**StarFS 10 replicas**
Directory Writes	0.225	0.441	1.176	2.135
File Writes	0.608	5.22	15.524	22.538
File Reads	0.4	1.049	1.803	1.968

Figure 3: Starfish Performance on Modified Andrew Benchmark in seconds (small files)