Makhzan

Omar Abdala, Basel Al-Naffouri, Faisal Anwar, Bilal Kaleem
{ota,naffouri,fanwar,mbilal}@mit.edu

December 6, 2002

Abstract

Scalability and performance are important design considerations for distributed file systems. Makhzan seeks to achieve these goals by reducing the responsibilities of the system's stable storage facility. Namely, we move responsibility for lock and cache management from the storage facility to the distributed file servers. We have implemented a scaled-down version of Makhzan and our performance results are promising - we conclude that future optimizations, as discussed in the paper, would allow Makhzan to perform on par with, and in some cases, better than the local file system.

1 Introduction

Distributed file systems (DFSs) consist of clients, servers, and storage. By client we mean an application that wishes to access data from the filesystem while servers process client requests. Stable storage is the back end of the DFS where the data is stored. Finally, by distributed, we mean that any part of the system (clients, servers and/or storage) can be dispersed over a network. Levy [3] discusses filesystems in more detail presenting examples such as NFS, AFS, and Sprite.

Figure 1 illustrates the main components of a DFS. It is meant to clarify the terminology we will be using for the different parts of a DFS but is by no means a description of all filesystems. For example, NFS combines and centralizes the server and storage layers rather than having them separate and distributed.

There are some basic requirements that a DFS should meet to provide useful semantics for client applications. These requirements may vary and so below we list those requirements that were considered during the design of Makhzan.

- The system must be network transparent. Namely, clients should access remote files using the same set of file operations as are used to access the local filesystem.
- Performance is a consideration, i.e. the system should not take a very long time to satisfy requests.
- The system should be fault tolerant to some degree. It should continue to function (albeit in a degraded state) despite some failures within it.
- The system should be scalable. Large numbers of clients should be served gracefully.

Makhzan attempts to satisfy all the requirements listed above but focuses specifically on scalability and performance. In the design section, we shall discuss network transparency (Makhzan provides a standard filesystem interface) and fault tolerance (a minimum set of guarantees in the face of crashes). But in the following section, we discuss the two main considerations - we discuss the scalability and performance limitations of different DFS architectures and motivate the architecture for Makhzan.

1.1 Performance and Scalability Considerations Within File Systems

The most naive architecture for providing distributed access to a filesystem is a single filesystem server that can be accessed by many clients. NFS [5] is an example of this approach. One server communicates with a set of clients. The obvious drawback is that it requires one node to manage all client requests and storage operations. Such configurations have a natural limitation in scalability and performance because one machine has to manage all requests.

A second approach is a multi-server model with a single centralized storage device. An example of

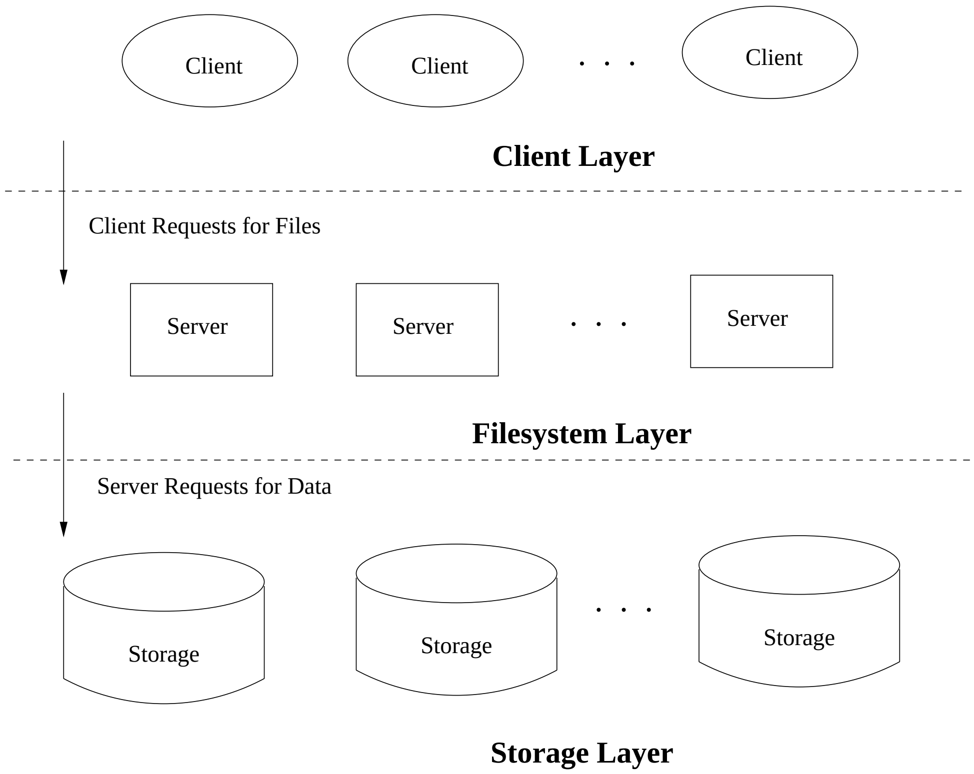

Figure 1: Distributed File System Layers

such a system is Frangipani [6] connected to a single Petal [2] virtual disk. In this model, client requests are multiplexed across several servers. However, storage is still centralized at one entity thus the storage layer has to be involved in any operation that changes the state of the filesystem (ie. changes to file data or inodes). The resulting performance slowdown can be alleviated through caching schemes (as is done by Frangipani). However, such file system designs still have the basic problem that a central entity (the storage device) is responsible for a vast amount of functionality within the DFS (including storage, and managing locks and copies of data). Indeed, as Anderson et. al point out [1], centralized modules within distributed filesystems are a major performance and scalability bottleneck. Thus, it is useful to consider approaches in which some or all of the load on a centralized entity is removed.

Accordingly, the approach that Makhzan uses is to multiplex client requests across many servers and to move some of the responsibility that traditionally rested with the storage layer on to the server layer. The storage layer is still responsible for non-volatile storage of data, but copies of the data can be stored within the server layer. Server layer machines can communicate with each other through a LAN and share copies of data without always needing to communicate through the central storage layer. Reads and writes are thus done primarily in the server layer, with periodic write back to the central storage layer. On the other hand, a system such as Frangipani [6] does not allow communication between server layer machines and therefore must use the storage layer to communicate locking between server layer machines.

The rest of this paper is arranged as follows. Section 2 presents the design of Makhzan. Section 3 discusses the modules of Makhzan that have been implemented. Section 4 discusses performance results. Section 5 discuss related work.

2 Design

2.1 Design Overview

Figure 2 presents a high-level picture of the makhzan filesystem. Clients, the top layer, seek to access some set of files residing on the file system. Meanwhile, the Makhzan layer updates and serves all requests for file metadata. Clients talk to

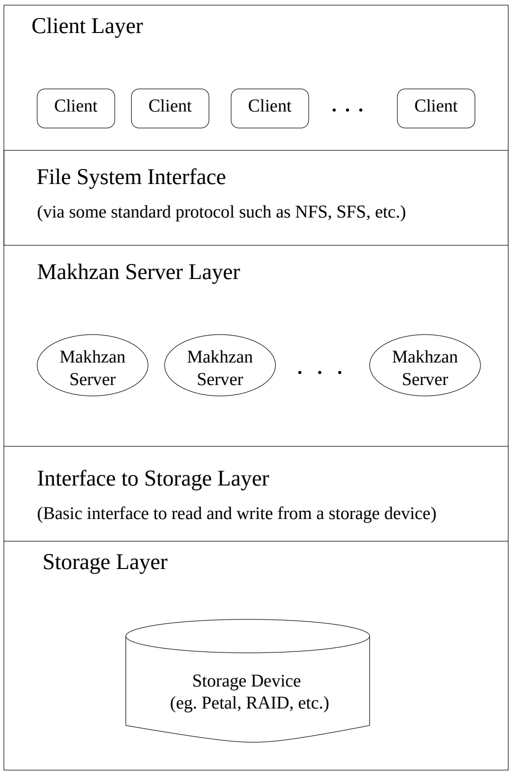

Figure 2: Makhzan System Overview

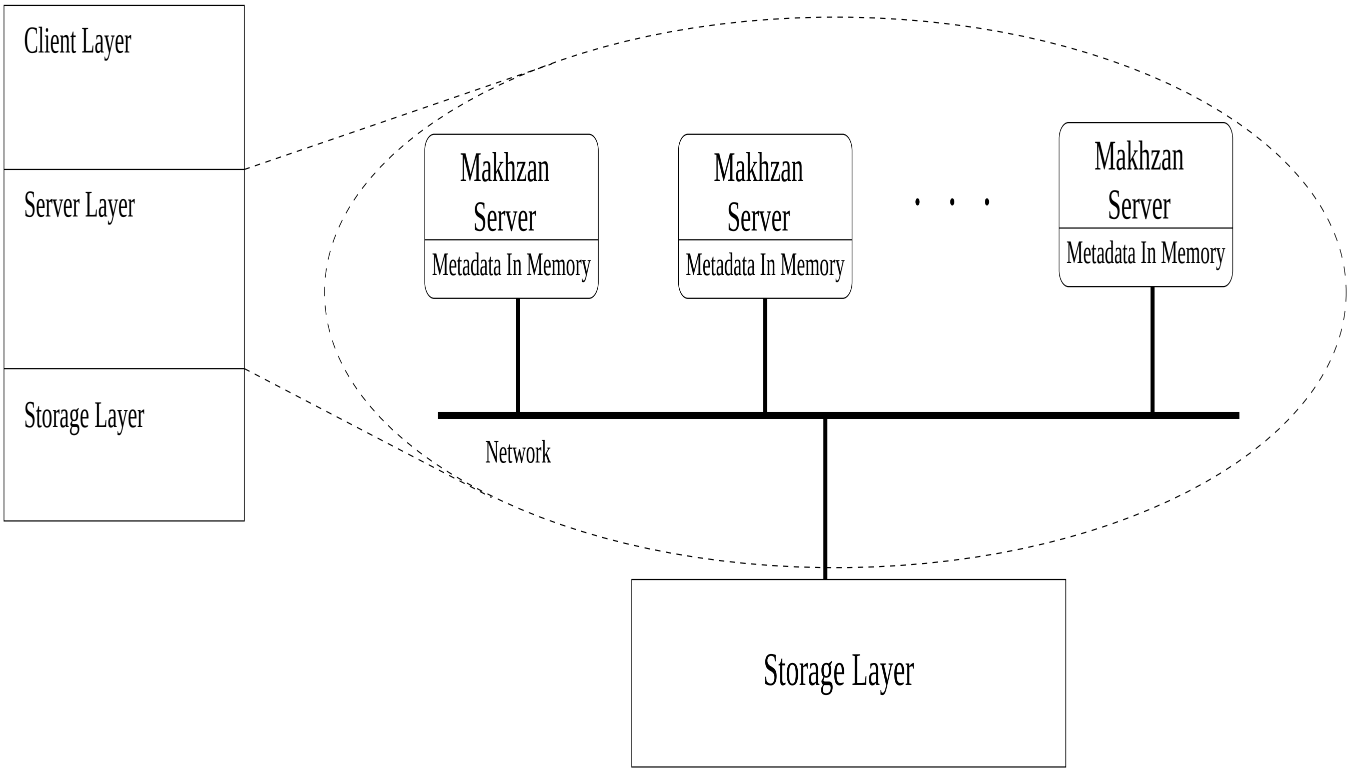

Figure 3: Server Layer

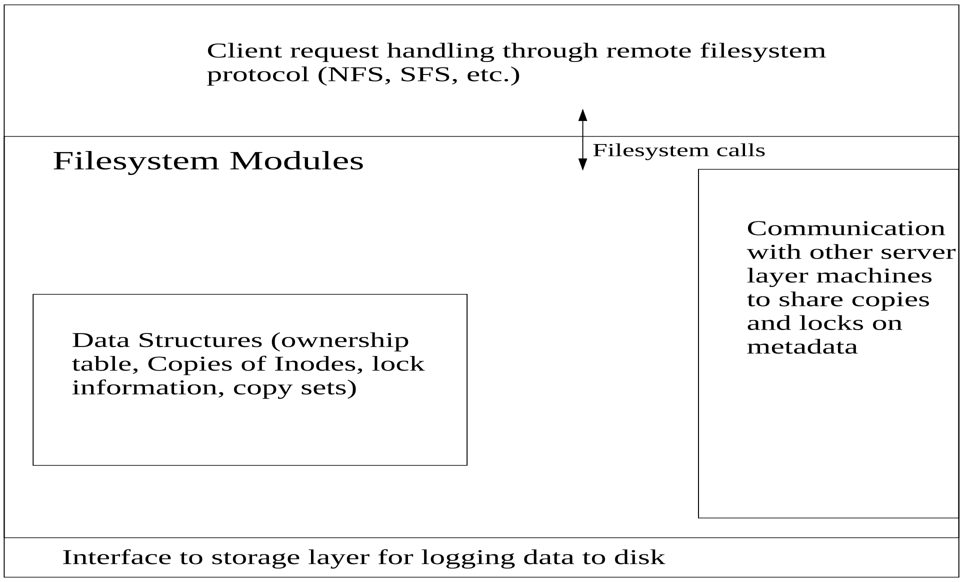

Figure 4: Individual Server Within the Server Layer

the Makhzan layer through some network file access protocol (for example, NFS, SFS, etc). The stable storage is an abstraction of a large stable storage (such as PETAL, RAID, any big disk, etc). This stable storage contains the actual data, a log of metadata operations, and a copy of all the metadata in a data structure that is the final authority.

Next, we look more closely at the Makhzan layer (see figure 3). The heart of the Makhzan layer is the set of Intermediate Servers (IS). We use multiple ISs to achieve fault tolerance (see 2.5) and scalability (as previously discussed). The ISs store all file metadata in memory and serve it upon request. Makhzan's definition of metadata is not limited to inode information. A reasonably sized data cache, the entries of a directory, or any other relevant file information can be included in metadata. This means that the performance gain achieved by Makhzan can be increased by adding to the "metadata" any particular part of the file that is in high demand.

Finally, we look closely at a single IS (see figure 4). An IS consists of three parts: the interface through which the network file system protocols talk to the IS, the facility for inter-IS communication (see 2.2), and the interaction with stable storage (see 2.4). Each IS owns the metadata for a set of files. Only an IS that owns the metadata for a file can change that metadata - by locking the metadata, and then writing it. An IS that wants to change metadata for files it does not own must first obtain ownership. An IS that simply wants to read metadata may request a read copy from the owner. Makhzan deals with IS crashes and maintains a consistent filesystem by logging metadata to stable storage.

The key difference between Makhzan and a system such as Frangipani [6] is that Makhzan's servers communicate with each other, while Frangipani's do not. The advantage to this approach is two-fold: (a) it removes the need for communication with the storage layer when obtaining locks and (b) it reduces the number of operations involved in a typical "request lock" operation. For example, in Frangipani, a server requesting a lock needs to: (1) ask the lock server at the storage layer for the lock, (2) the storage layer has to ask the server holding the lock to give up the lock, (3) the server with the lock has to write its metadata to storage and yield the lock and (4) the requesting server must then take out the lock and read the current data that was just written. Meanwhile, in Makhzan, the servers can negotiate lock exchange on their own without the involvement of a lock server (we implement a distributed lock service) and without the involvement of the storage layer (in the common case). Thus, a server cuts the time needed to get locks on metadata because there is no need for synchronous communication to the storage layer.

2.2 Inter-IS Communication: Metadata Locking and Transfer

This section discusses how metadata is stored, accessed and shared in a distributed manner by the

IS layer. First, we present the data structures that each IS uses to maintain the metadata. Next, we present a description of the algorithms for various inter-IS communication and their pseudocode. These data structures and algorithms build upon IvyDSM [4].

The data structures at each IS are:
inode-table: the inodes for all owned files.
probable-owner-table: the "probable owner" for each inode.
own-set: set of inodes this IS owns
copy-set (defined per inode this IS owns): set of ISs that have a copy of the metadata
read-set: set of inodes this IS has a copy of and can read from
lock-set: set of inodes this IS has locked (for a write)

In order to simply read metadata (not write), an IS needs a copy of that inode. The following is a description of the interaction between an inode requester, and the inode owner as described in the two pseudocode functions requestCopy and giveCopy (see 2.3). The IS requester first checks if it owns the inode. If it does, it reads it. Otherwise, it checks whether it knows of a probable owner. If yes, it asks that machine for a read copy. In return, it may receive one of four replies:
(1) INODE_RETURNED: means that a copy of the metadata has been sent and that the inode owner has added the inode requester to the copy set of that inode.
(2) INODE_UNAVAILABLE: means the machine contacted *does* have the metadata but is locked. The resulting action will be a retransmission of the request after some wait period.
(3) OWNER_KNOWN: means that the machine contacted does not own the metadata, but it returns its notion of the probable owner. The requester updates its probable owner field and retransmits the request to the new probable owner.
(4) OWNER_UNKNOWN: means the IS contacted does not have the inode nor knows of any other probable owner. Thus the inode requestor cannot reach the owner using the probable owner fields, and therefore must broadcast its request to all ISs. The owner will then recieve the request and respond accordingly.

In order to write metadata, an IS needs to obtain ownership and lock that inode. The following is a description of the interaction between an ownership requester, and the owner, as presented in the two pseudocode functions requestOwnership and giveOwnership. When an IS tries to acquire a write lock, a similar sequence to the one on read occurs. If the IS is the owner, it invalidates its copy set (notifies all copy holders that their copy is invalid), sets the lock for the metadata, and writes. If the IS is not the owner, it checks for a probable owner. If one exists, it asks that machine for a write lock. The same responses to the read request exist for the write request:
(1) On INODE_RETURNED, the requester will add the metadata to its inode-table, make itself the owner, and lock the metadata. The machine giving up the metadata has now invalidated its copy set.
(2) On INODE_UNAVAILABLE, wait and retransmit request later (as above).
(3) On OWNER_KNOWN, retramsmit to the new probable owner (as above).
(4) On an OWNER_UNKNOWN or when there is no probable owner, the requester must broadcast its request for a write lock. The owner will then respond and give the IS the write lock as in the normal case.

That is the simplified big picture. There are several other issues to discuss. These will be discussed in 2.4 and 2.5.
S

2.3 Metadata Algorithm Pseudocode

Here we present the psuedocode for the algorithms described in subsectionMetadata Locking and Transfer Between IS Machines.

```
requestCopy(inode#) {
  owner = probOwner[inode#];
   if(owner) {
     [inode, return_val, owner] = rpc_giveMeCopy(inode#, owner);
     if(return_val == INODE_RETURNED)
        return inode;
     else if(return_val == INDOE_UNAVAILABLE)
        return retry_delayed(requestCopy(inode#));
     else if(return_val == OWNER_KNOWN) {
        probOwner[inode#] = owner;
        reuturn requestCopy(inode#);
  }
     [inode, owner] = rpc_broadcastRequestCopy(inode#);
     if(!inode) {
        inode  = assumeOwnership(inode#);
     } else {
        probOwner[inode#]=owner;
     }
     return inode;
  }
}

giveCopy(inode#) {
  if(probOwner[inode#] = me) {
    if(locked[inode#])
       return [NULL, INODE_UNAVAILABLE, me];
    else{
       copySet[inode#].add(requester);
       return [inodes[inode#], INODE_RETURNED, me];
    }
```

```
  } else if(probOwner[inode#])
      return [NULL, OWNER_KNOWN, probOwner[inode#]];
    else
      return [NULL, OWNER_UNKNOWN, NULL];
}

requestOwnershipAndLock(inode#) {
  owner = probOwner[inode#];
   if(owner) {
     [inode, return_val, owner] =
        giveMeOwnership(inode#, owner);
     if(return_val == INODE_RETURNED) {
        probOwner[inode#] = me;
locked[inode#] = true;
        return inode;
     }
     else if(return_val == INDOE_UNAVAILABLE)
        return retry_delayed(
          requestOwnershipAndLock(inode#));
     else if(return_val == OWNER_KNOWN) {
        probOwner[inode#] = owner;
        reuturn requestOwnershipAndLock(inode#);
     }
     [inode, owner] = broadcastGiveMeOwnership(inode#);
     if(!inode) {
        inode = assumeOwnership(inode#);
     } else {
        probOwner[inode#]=me;
     }
     return inode;
  }
}

giveOwnership(inode#) {
  if(probOwner[inode#] = me) {
    if(locked[inode#])
      return [NULL, INODE_UNAVAILABLE, me];
    copySet[inode#].nullify();
    inode = inodes[inode#];
    inodes.remove(indoe#);
    return [inode, INODE_RETURNED, requester];
  } else if(probOwner[inode#])
      return [NULL, OWNER_KNOWN, probOwner[inode#]];
    else
      return [NULL, OWNER_UNKNOWN, NULL];
}

assumeOwnership(inode#) {
  broadcastIAmOwner(inode#);
  probOwner[inode#] = me;
  inodes[inode#] = readInodeFromDS(inode#);
  return inodes[inode#];
}
```

2.4 Logging

In order to maintain a consistent file system which deals with metadata updates atomically in the face of IS crashes, Makhzan maintains a "final authority" on the metadata and a log, both of which reside on stable storage. Any operations which update metadata result in all changed metadata being asynchronously written to the log as a single log entry. The log is a series of atomic operation records. Each record consists of a start message, all the updated metadata, and a commit message (as shown in figure 5.)

Each IS has a list of log entries that it maintains, and an inode flush list. If a metadata change is made in an operation, the inodes are added to the inode flush list. Also, the log entry is put on the local log. The log entries on each IS are asynchronously flushed to stable storage as frequently as possible. Upon receiving a request to give up ownership of an inode, an IS first checks if that inode is in the flush list, and if so, flushes its log entries to disk before yielding ownership. This guarantees correct ordering of filesystem operations because an IS is required to have updated the final authority of any metadata changes the IS has made before a new IS can write that inode's metadata. Atomicity of metadata operations is guaranteed by the fact that all metadata changed by an operation is in one log entry.

It is our expectation that the IS will be able to update the log far more frequently than it will get requests for ownership from other ISs. Therefore, the likelihood of an ownership requester having to wait for an owner to log flush before returning is small. Thus, the log flush, a necessity for filesystem correctness is, on average, a negligible latency hit. Throughput experirnces a negligible hit, since the small amount of data that the ISs will be writing to the stable storage will not be comparable to the amount read and written due to normal filesystem operations.

As the log grows, compressing the log is required to save time and space. The log will have redundant information, which will slow down the linear log searches and take up more space than necessary. Compression works by making a copy of the log and reading through that copy, forming a data structure with the most up-to-date metadata. This structure is a snapshot of the file system state - a new "final authority". This new final authority replaces the old final authority. The old log (up to the last entry that was copied) can now be removed, and the log space reused.

When a request to the storage layer asks for the most recent copy of the metadata, the storage layer searches backward through the log and returns the first inode it finds. If it has reached the beginning of the log and it has not found the inode, it will lookup the inode in the final authority and return that. As long as the compressing program is run frequently enough, log searches will not be time consuming.

2.5 Crash Recovery

After an IS crashes, several issues need to be resolved. First, inodes which were owned by the crashing IS have no owner anymore. Second, the probable owner table of the crashing IS is lost, so some of the probable owner links are disconnected. Third, the

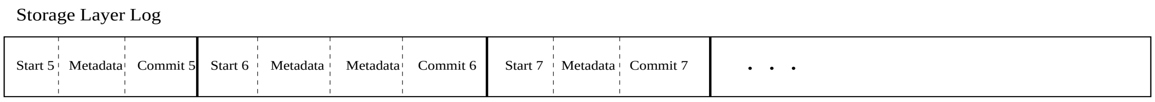

Figure 5: Log

file system needs to cope with consistency issues, such as ensuring atomic operations and the order of these operations. Finally, some file system operations might be lost due to the crash. We discuss the severity of such lost operations.

Makhzan implements a distributed crash recovery mechanism. After an IS crashes and boots up, it does not need to do anything special - it can immediatly begin serving requests to the clients. The IS slowly obtains ownership and read copies of inodes. All the machines in the IS layer recover from the crash by doing their usual operations. Thus, special crash recovery mechanisms are not needed.

After a crash, all the inodes that are owned by the crashing IS no longer have an owner. The Makhzan system copes with the lost ownership problem with the assumeOwnership operation (see section 2.3). In short, if a requesting IS cannot determine the owner of an inode using the probable owner links, the IS sends a broadcast requesting ownership. If no other IS is the owner, the IS simply assumes owenership of the inode. This is done by reading the inode from the log and sending out a broadcast claiming ownership. The broadcast also updates other IS's probOwner table. This alleviates the problem related to broken links in the probable owner chain. Instead of every IS having to broadcast to find the owner, we use the "I am owner" broadcast so that only one broadcast is needed to fix the broken links to this inode, rather than having as many broadcasts as ISs.

Another result of an IS crash is loss of the probable owner table. ISs that are linked to the real owner through the crashing IS no longer have a chain to that real owner. When an IS gets a OWNER_UNKNOWN or no reply from a probable owner, the requesting IS broadcasts to all IS's asking for the real owner to respond. The actual owner then replies to the requesting IS, and the operation continues as usual.

The third issue resulting from an IS crash is file system consistency. There are three points in time when the file system can crash: before the operation is complete on the IS, after the operation is complete but before the entry is written to the log in stable storage, and after the operation is complete and has been written to the log. If the system crashes in the middle of an atomic operation, that operation is not recorded and thus has no effect on the file system. If the operation completes but the IS crashes before the write to the log completes, then the operation is still not recorded and has no effect on the file system. The only time file system changes are recorded is after an operation is complete and the write to the log was successful, in which case the change is recorded correctly.

The fourth and final issue of concern is lost filesystem operations. Due to the asynchronous nature of log writing, some operations, which were finished on the IS, might not be written to the log before the crash occurs. Since the writes to log are done as frequently as possible, the number of lost operations is very small.

3 Implementation

3.1 Current Implementation

The current implementation of Makhzan is a scaled down version of the design discussed throughout section 2. We chose not to implement the logging mechanism or the crash recovery. We focused mainly on implementing the IS metadata sharing. This was done because the most interesting ideas are in the inode-sharing. Furthermore, the performance of the current implementation (i.e. no logging or crash recovery) should be comparable to the performance of the full design during normal operation (i.e. without crashes).

Makhzan's implementation was divided into two layers, the inode_handler and the Makhzan file system layers. The inode_handler layer is responsible for ownership operations, which are: obtaining and releasing ownership of inodes, obtaining and giving out inode read copies, and local locking to prevent ownership release. The Makhzan file system layer uses the inode_handler to implement file system operations such as rename and copy.

Makhzan's implementation is non-blocking, i.e. we provided an asynchronous API for Makhzan. We used the asynchronous library's (libasync's) callback procedure to acheive that goal. The result is an interface with filesystem calls (read, write, create, re-

Operation	Makhzan	EXT
stat	166.071	14.757
ls	247	85.463

Table 1: Time to coplete stat and ls on the local filesystem vs. Makhzan. (microseconds)

Operation	Makhzan	EXT
ls	1.686	1.075

Table 2: Time to complete ls operation that accesses the hard disk vs. a Makhzan ls operation that accesses the netowrk. (milliseconds)

name, unlink, etc.) which take callback functions to be called back when Makhzan returns with the appropriate function completed. It would be simple to add a synchronous interface on top of this one by initiating a call to Makhzan, waiting, and returning upon callback.

ISs communicate using three RPC calls. The first is a request to give ownership of an inode. The second is a request to give a copy of an owned inode. The third is a request to nullify the copy of a given inode.

Finally, those who are familiar with IvyDSM [4] may wonder why we did not simply use Ivy. Ivy is a system for managing a shared memory over multiple machines. Makhzan is similar function except that Makhzan operates at the level of inodes while Ivy operates at the level of memory pages. We do not use Ivy for a couple reasons. First, we need to handle the case where some machines in the system have crashed while Ivy does not deal with crashes. Second, we needed to implement a concept of inode locking such that ownership is not given up in the middle of a write. We would have had to make a DSM system compatible with this before we used it. Therefore, we opted to implement shared inode management ourselves but still relied on much of the algorithms presented in Ivy.

3.2 Future Work

In order to fully test the Makhzan idea, we would need to extend our implementation in several important ways. First, we would need to implement the flush list and operation logging on ISs. Second, we would need to write a daemon that, in the background, would flush the local log to stable storage and clear the flush list. There must also be an interface for synchronous requests from the inode_handler for a log flush. Third, we would need to implement a function to assume ownership as described in section 2.5. Fourth, we would need to implement the cleanup program described in section 2.4 for the log in stable storage.

4 Performance

To evaluate Makhzan's performance, we ran some basic filesystem operations on the EXT filesystem (used on Redhat Linux) with a locally mounted disk and compared it to the same operations run on Makhzan. The numbers were gathered on ISs running Makhzan over an Ethernet network. The machines were close to each other, contributing to very fast network speeds. Such network conditions are a reasonable assumption for any environment where we anticipate Makhzan to be used.

We compared stat and ls on the local filesystem to those on a Makhzan file whose inode is in memory. The results of this comparison are in table 1. We also compare an ls operation that needs to access the hard disk to a Makhzan ls operation where the IS needs to request the inode from another IS over the network. The results of this comparison are shown in table 2. All operations were performed 1000 times and the results were averaged to produce the values in the tables.

With reference to table 1, there are several reasons why EXT can do these operations faster than Makhzan. Firstly, the filesystem caches the operations performed while Makhzan does not. Therefore, it performs the operation again every time. Secondly, our system uses asynchronous callbacks to improve the overall throughput of the system. In the case where all information required to carry out the operation is local in memory, this is very wasteful. It incurs double the procedure calling overhead for no gain. It would be simple to implement a check for residency of an inode in the IS. Based on this

type	requestOwnership	requestCopy
local	166.071	194.59
remote	1686	1742

Table 3: Time to complete requestOwnership operation vs. requestCopy operation when the inode exists on local and remote machine. (microseconds)

check, the client would either call the asynchronous interface with callbacks, or call a synchronous interface which does not use callbacks. Thus it would not incur the performance penalty in the case where the request can quickly be synchronously served. Thirdly, we have spent no time optimizing our code. We expect that we could get the performance of our system on resident inodes almost down to that of EXT if we invest in code and data structure optimization.

With reference to table 2, the important thing to note is that accessing inodes from machines on the network is faster than accessing them on the local disk. We can assume that the forced log flushes to stable storage are relatively infrequent. We can further assume that the probability of having to go to the network for an inode is, at maximum, only slightly greater than that of having to go to disk and that the probability of an IS crash is small. These assumptions imply that Makhzan can run at least as fast (on file information included in metadata) as the local filesystem.

Overall, Makhzan's performance was inhibited by shortcomings in the implementation but the tests also encourage us to conclude that Makhzan has the potential to perform very well if if we work on the various implmentation flaws discussed.

5 Related Work

Distributed File Systems can have many different architectures. NFS [5] is a filesystem protocol that defines a set of operations clients can request from a remote server. NFS, too, can have a couple different architectures. For example, a single, centralized storage/server machine versus a many-server/single-storage-device configuration. Both configurations have major limitations because the centralized storage is responsible for too much within the filesystem. This point is obvious in the first configuration but it is also true in the second configuration: the NFS servers machines can only communicate with each other (for locking and caching purposes) by going through the central NFS storage server.

Frangipani [6] is a file system that uses a distributed set of servers talking to a single "virtual disk" (storage layer). One of the key features of Frangipani is that servers do not talk with one another and therefore locking of cached data cannot be implemented without major involvement from the storage layer. Newer versions of Frangipani do implement a distributed lock service, which results in a structure similar to that described for Makhzan.

XFS [1] is a file system that takes the notion of decentralizing distributed file systems to the extreme - it removes any central storage entity. Storage and client service are all implemented on a distributed set of machines that talk to one another in order to provide availability of all data at any server. Thus, client requests are multiplexed across these machines as well as data storage responsibilities.

Distributed shared memory systems, such as Ivy [4], implement a distributed memory in much the same way we implement distributed data caching. Indeed, Ivy provided us a good starting point when deciding how to manage copies of data across many servers and how to implement read and write locking.

Acknowledgements

We thank Robert Morris for his help in directing and focusing our thoughts and pointing out places where we could improve our design. We also thank the rest of the 6.824 staff for their help in some debugging on our implementation that otherwise would have been quite time consuming.

References

[1] Thomas Anderson, Michael Dahlin, Jeanna Neefe, David Pat-terson, Drew Roselli, and Randolph Wang. Serverless network file systems. In *In Proceedings of the 15th Symposium on Operating System Principles. ACM*, pages 109–126, Copper Mountain Resort, Colorado, December 1995.

[2] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, Cambridge, MA, 1996.

[3] Eliezer Levy and Abraham Silberschatz. Distributed file systems: concepts and examples. In *ACM Computing Surveys (CSUR)*, pages 321 – 374, December 1990.

[4] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. In *Proc. 5th Symposium on Principals of Distributed Computing*, pages 229–239, August 1986.

[5] Steve Kleiman Russel Sandberg, David Goldberg, Dan Walsh, and Bob Lyon. Design and implementation of the sun network filesystem. In *Proc. of USENIX 1985 Summer Conference*, Portland, OR, Summer 1985.

[6] 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.