# FSFS: Fault-tolerant Secure File System

**Kaushik Chowdhury**
**Justin Fiore**
**Erich Stuntebeck**

School of Electrical & Computer Engineering
Georgia Institute of Technology, Atlanta, GA 30332
Email: {kaushikc, eps}@ece.gatech.edu, justin.fiore@gatech.edu

# Contents

**1 Introduction** **1**

**2 Related Work** **1**

**3 System Design** **3**

3.1 FSFS architecture . . . . . . . . 3

3.2 Underlying Storage Device . . . . . . . . 4

3.3 Database Structure . . . . . . . . 5

3.4 Fault Model and Fault-Tolerance Mechanisms . . . . . . . . 5

3.5 Security considerations . . . . . . . . 5

**4 Implementation Details** **7**

4.1 Architecture of the JINI platform . . . . . . . . 7

4.2 MasterService Recovery . . . . . . . . 8

4.3 Implementation Details by Example . . . . . . . . 8

4.4 Deployment . . . . . . . . 9

4.5 Issues Encountered and Addressed . . . . . . . . 10

**5 Results** **10**

5.1 Simulation setup . . . . . . . . 10

5.2 Observations and discussion . . . . . . . . 11

**6 Conclusion** **12**

# 1 Introduction

For this project, the authors set out to design a secure, fault-tolerant, platform-independent, distributed file system. Our file system is designed to replicate stored data amongst the machines participating as servers in the file system. Replication of data provides both increased reliability (should one or more servers fail data can be recovered from other servers in the system which are still functional) as well as increased performance (clients can fetch data replicated on several servers from the one with the lowest latency). The goal was to create a system meeting all of these requirements and to evaluate that system on several fronts against existing solutions in this space.

Along the way, new ideas and discoveries prompted us to reevaluate our initial design, and thus the final implemented system differs somewhat from the initially submitted proposal. Initially, it was proposed to use a ring-based approach along with a consistent hashing algorithm for placement of data throughout servers in the system. This approach was abandoned however, in favor of a an approach incorporating a ”master server” which contains references to all data in the system. Although it may seem that this new approach introduces a single point of failure to the system, our design alleviates this concern by incorporating the capability to reconstruct the metadata contained by the master server on any individual server in the system should the master server experience a fault. Additionally, this allows the file system service, which acts as the interface of the system to clients, to quickly locate data in the system by contacting one source. Details of this approach are provided in the following sections.

Our design consists of four primary services, which may be distributed throughout physical machines as desired: the block service, the file system service, the master service, and the client. The client is, of course, the end user application that desires to take advantage of the services provided by FSFS. The client contacts the file system service, which in our setup was run on the same machine as a block service. The file system service is responsible for all interaction with the client and services the client's requests by contacting the master server and other block servers as appropriate to store or retrieve data. The master service, of which only one entity exists within a FSFS system, is responsible for storing metadata on all blocks (pieces of a file) stored within the file system.

FSFS is designed to be a secure file system. Security encompasses several aspects. First, data must be secure in transit across an un-trusted network such as the Internet. This is accomplished through the use of transport-layer security provided by SSL. Additionally, data must be secure while stored on the disk of whatever block server it eventually resides on. This security is provided through a combination of AES encryption (to preserve the privacy of the data against an un-trusted operator of the machine on which the block service operates) and SHA-256 hashes (to verify integrity of the stored data against adversaries that wish to modify it or against bit errors due to hardware failure).

The remainder of this paper is devoted to explaining the design and functioning of FSFS in detail. Additionally, we present our methodologies for evaluating FSFS and the results of those evaluations. The final state of the system is then compared to the initial goals for the project.

# 2 Related Work

There has been an active research thrust on devising improved methods for organizing, storing and manipulating files, beginning from single computer based approaches and to the more advanced distributed file systems. *New*

*Technology File System* (NTFS) [1] for Windows based machines and the *third extended filesystem* (ext3) [2] for Linux are examples of the former where all file operations are performed locally by the user machine. We will, however, focus on distributed file systems that allow sharing of files between multiple users within a network. We restrict the scope of this discussion to software implementations that interact with the operating system (OS) and the physical storage devices that is spread over a variety of different machines. The reader is encouraged to refer to [3] for a comprehensive treatment of current distributed file system technologies and we focus mainly on contrasting our approach with some of the available systems.

Coda [4], devised by CMU, was originally designed for UNIX systems but now has been ported to Windows95, FreeBSD, and Linux platforms. It makes a distinction between servers, which are relatively few in number, and clients, which are far more numerous. Coda replicates collections of files, and the set of servers that contain these replicas are called *volume storage group* (VSG). The membership of the servers to a given VSG is dynamic and the subset that is currently accessible is made known by the system at all times. The replication strategy used is a variant of the read-one, write-all approach. When a file is closed after modification, it is transferred to all members of the accessible subset. Coda minimizes server CPU load by placing the burden of data propagation on the client rather than the server. This in turn improves scalability, since the server CPU is the bottleneck in many distributed file systems. The key drawback, latency of synchronous propagation, is addressed in Coda by the use of a parallel remote procedure call mechanism. Similar to CODA, our FSFS system undertakes file replication and contains several policies that govern read/write access. The updating of the individual blocks stored in the system is done by the central master server on the basis of sequence numbers described in detail in Section 3.

The Windows Distributed File System (DFS) [5] is a new tool for the NT server that allows administrators to simulate a single server share environment that actually exists over several servers. This allows a common directory and file structure visible to the end user, thus simplifying backup and update operations as well. FSFS follows a similar architecture in which the entire database structure is available at a centralized location. The DFS system then takes responsibility of actually collecting the information from all the distributed caches containing the files.

Large scale distributed networks providing Peer-to-Peer (P2P) services like Napster [6], Gnutella [7], Kazaa [8] and Morpheus [9] have achieved a wide user base recently. Napster uses a centralized directory of object locations that proves to be a bottleneck. While our architecture has a centralized element in the form of a master-server, the system is completely recoverable in event of failure at this central point. However, like Napster, the directory structure is available at the central location and anonymity has not been a focus in our approach. Freenet [10] is an adaptive peer-to-peer file system that that protects the anonymity of the authors, data location and the readers. It uses probabilistic routing to preserve the anonymity of its users, data publishers, and data hosts and incorporates the goals of deniability for the storers of information, resistance to third party access, dynamic storage and routing, and decentralized policy [3]. Not withstanding the related legal problems, P2P systems that offer a decentralized, self-sustained, scalable, and fault tolerant architecture seem to hold considerable promise in the future.

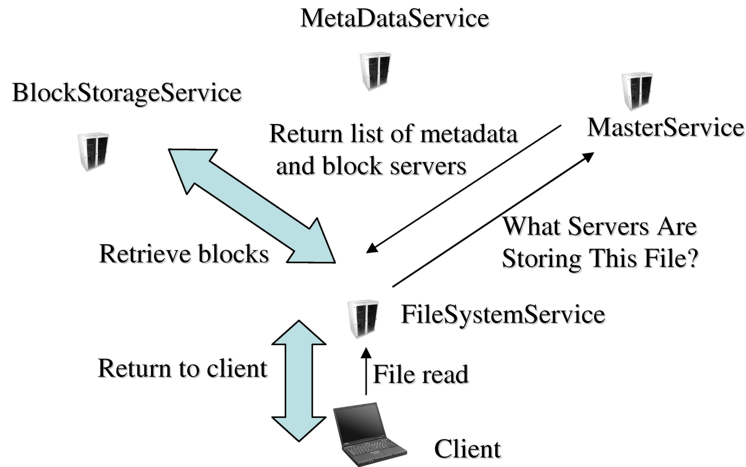


Figure 1: A block diagram showing the interaction between the various services. The client requests a file from its local server which then forwards the request to the master server. This returns the locations where the individual blocks are stored. These blocks are retrieved by the local server and passed on to the client.

Table 1: Descriptions of the services provided by FSFS

| Service Names | Service Description |
|---|---|
| MasterService ($MS$) | Service that aggregates all metadata and handles any atomic operations |
| Lookup Service (Reggie) ($LS$) | Service that aggregates all proxy objects and is used for lookup of available services |
| BlockStorageService ($BS$) | Service that provides block storage and retrieval mechanism |
| MetaDataService ($MS$) | Service that provides metadata storage and retrieval mechanism |
| FileSystemService ($FS$) | Service that provides file system operations |

# 3 System Design

## 3.1 FSFS architecture

The overall design of FSFS is using a service-oriented architecture (SOA). Service providers provide services to clients, shown in Figure 1. Any process could potentially be a client and/or a service. In this way, services can utilize other services in order to accomplish their tasks. A service is defined by its interface, which is simply a list of methods that can be invoked upon it. The services in FSFS are defined as Java interfaces, because Java is our implementation language and platform. A short description of each service can be found in Table 1

The FSFS architecture was designed to avoid the need for atomic multicasts, since guaranteeing this across a wide range of systems is very difficult at best, and can be intractable. To this end, a system was envisioned in which there would be some centralized services provided, but there would never be a single point of failure. FSFS uses a single MasterService that is responsible for any actions that require any degree of atomicity, such as adding and removing files and directories, acquiring locks for files and several other functions. Also, the MasterService is responsible for providing services that require full knowledge of the system, such as file or directory listing, file opening, etc. The reason for this is that the MasterService is the only system that has the

knowledge of the full system and therefore the MasterService is always up-to-date because if it does not know about a file or directory, it doesn't exist in the current makeup of the filesystem.

However, we must treat the MasterService as a transient service. By this we mean that when the MasterService crashes, a new MasterService can be created by querying the other services in the system. The only downtime the system will experience will be in the interim between when a MasterService crashes and when a new one has finished initializing. The downtime denies service to those who want to use the filesystem but it keeps the filesystem in tact, thus making the system in general reliable, even if up-time is not 100%.

The system utilizes three other services besides the MasterService: the FileSystemService, the BlockStorageService, and the MetaDataService. The FileSystemService is the service that the client of FSFS would use. The services provided by the FileSystemService is all the client sees of the system. The FileSystemService uses the MasterService, the BlockStorageService, and the MetaDataService. The MetaDataService is responsible for storing, updating, and providing the metadata of the filesystem. Also, the MetaDataService is responsible for starting a new MasterService when the MasterService crashes. Lastly, the BlockStorageService is responsible for receiving, storing, and providing individual blocks of files on the filesystem. Except for the MasterService, there can be any number of the these services. The use of multiple FileSystemServices could be used to distribute load and also to provide FileSystemServices that are geographically closer than another FileSystemService. The reason for using multiple MetaDataServices and BlockStorageServices is to distribute the load of the filesystem and also to provided data-level replication for files on the filesystem and also the filesystem structure itself, the metadata.

The last major high-level design concern is how to handle concurrent file accesses. The FSFS filesystem is strict in the sense that every successful open creates a file lock. We allow a file to be read simultaneously by multiple clients, and to be written by only one client. A write lock (or a read/write lock) is an exclusive lock such that no other client may access the file concurrently. In a faulty environment this could be dangerous, as locks for crashed clients could be held indefinitely. FSFS was designed to use a leasing concept for locks. When a successful file open occurs, a lock is granted with a time that it is set to expire. The client should periodically renew the lease for continued use of the lock. If a lock expires, the lock is removed and that client must reopen the file in order to access it again. This guarantees that in the event of a crash, a file is unavailable to be opened due to locking for at most one lease cycle.

### 3.2 Underlying Storage Device

Typical non-distributed filesystems are implemented on top of block devices. In that regard, each metadata record, or inode, is stored within one block. Files can span multiple discontiguous blocks. Our implementation does not use an underlying block storage device, but instead uses the services provided by the local non-distributed filesystem to store blocks and metadata. The metadata is stored in embedded relational databases that are stored on the local filesystem. Individual blocks of data are also stored on the local filesystem. Each file has its own directory and in that directory, there is a file for block corresponding to that file that a given server is storing.

This was done not only for simplicity of implementation, but also for simplicity of deployment. There is no formatting of a filesystem needed to deploy the distributed filesystem. The only thing needed in terms of storage requirements is a directory with read/write access and some disk space.

### 3.3 Database Structure

The metadata for the filesystem as well as metadata about individual blocks is stored in several database structures. The database system used was Java DB, now called Derby. This was chosen because it an easily embeddable relational SQL database that we could include easily with the services that needed a database.

There are three databases: masterDB, metadataDB, and blockDB. MasterDB is a superset of metadataDB and blockDB containing all tables from both databases, but also containing extra columns to keep track of on which server individual blocks and pieces of metadata are stored and to keep track of locks that are presently being used. The masterDB is generated from scratch each time a MasterService is initialized. The masterDB's data ultimately is the aggregate of all the metadata and block information in the system.

The metadataDB contains the following tables: metadata, directory_metadata, file_metadata, indirect_inode, and users. Metadata stores general metadata information such as: the ID, owner, group, creation and modification times, and permissions. The directory_metadata table contains information about a directory such as: the number of files in the directory, pointers to the other items in the directory. These pointers are just the ID of the appropriate record. The file_metadata table contains information about files, such as: file size and other file specific information. The indirect_inode table represents an indirect inode in a file system, which is an inode that contains more pointers to other inodes. This was designed so that there can be more than 10 files per directory. The directory_metadata table only has 10 pointers per directory because most directories may not have more than 10 files, so creating database space for a large amount of children nodes would be wasteful. Lastly, the users table contains usernames, password hashes, and groups that a given user is a member of. The default group is the first group listed.

The blockDB contains information about individual blocks. Specifically, it contains: the fileID, blockID, the filename where that block is stored on the local filesystem, and a hash used for integrity checking the blocks during read.

Each table in all databases also has a seq_num field for keeping track of which data is most current. When compiling the masterDB, the MasterService keeps only the records with the highest seq_num and informs out-of-date servers to get the up-to-date versions from other servers.

### 3.4 Fault Model and Fault-Tolerance Mechanisms

The fault model we assume with this system is a crash fault model. There is also support for verifying the integrity of blocks when they are read from the disk, but a true Byzantine fault model is not supported.

The fault-tolerance of the data and metadata in the system is achieved by replicating the data on r different servers, where r is the replication degree. If r distinct servers cannot be utilized, the data is stored on all available servers. The fault-tolerance of the MasterService is achieved by: detecting a MasterService crash, electing a new MasterService, starting the new MasterService, and notifying other services in the system of the new MasterService. After detecting that the MasterServer has crashed, the FileSystemService must pause any operations on the filesystem until the new MasterService is operational. The implementation of the MasterService recovery will be discussed in Section 4.2.

### 3.5 Security considerations

Security in FSFS consists of four components: transport-layer security, security of data stored on the servers, integrity of data stored on the servers, and access control for users of the system.

Transport-layer security is a necessary aspect of any distributed file system that claims to be secure. The content of data stored in the file system must be protected as it traverses un-trusted nodes on the network, where it could be intercepted and modified by an adversary. Thankfully transport-layer security is a well-explored topic, driven by the need for secure commerce over the global Internet. The underlying distributed communication architecture of our file system, JINI [11], supports the use of SSL (Secure Sockets Layer) for secure communication. We utilized this solution in order to provide security of data while in transit.

It is also important that data be secure while stored on the disk of each server in the distributed file system. Security here can mean both protecting the data from examination by those other than intended by its owner as well as the detection of any modifications to the data while stored on the server, which could be introduced by hardware and/or software failures, as well as an adversary. The simplest solution to this problem would be to encrypt data before it ever leaves the client machine, meaning that it is never stored on the servers in an un-encrypted manner. This would also provide a simple solution to transport layer security of data (not control commands however). The problem with this approach, however, is that a file system should be designed with multiple users in mind. A user may wish to share their data with other users of the FSFS system. Encrypting the data with a secret key known only by the owner of the file before it ever leaves the originating machine would clearly prevent this (unless the file owner shared this secret key with all intended users of the file, a cumbersome and undesirable solution to this problem). Given that we utilized a more flexible approach to transport-layer security (SSL), this gave us the option of not encrypting the data before it leaves the originating user's machine while still maintaining its security. To provide security on disk, data is encrypted using the AES encryption standard and a 256-bit key maintained by each block server. Upon receipt of a block, the block server encrypts the block with its key before writing it to the local disk. In this manner, a block server can be run on an un-trusted machine, so long as the operator of the block server application (who may have access to the block server key) is a trusted entity.

Although this method guarantees the privacy of data, data security also encompasses integrity. This implies that the data the user writes to the file system is the same data received when he or she requests that data again in the future. In order to guarantee this aspect of data security in FSFS, a cryptographic hash is utilized. A collision-free hash algorithm will provide a 1-to-1 mapping from a particular block of data to a hash code. We opted to utilize the SHA-256 algorithm. Although MD5 and SHA-1 are popular choices, both of these algorithms have been compromised with the discovery of collisions. Upon receipt of a block of data, the block server, in addition to encrypting this data, will also calculate the SHA-256 hash of the un-encrypted data and store this in its metadata table. In this manner, changes to the original data can be detected. For example, should an adversary modify the encrypted data, the hash of the un-encrypted data will also change. When reading back a block, the hash should be taken of the read data and compared with the hash stored in the metadata table to detect any changes.

The final aspect of security in FSFS is security amongst users of the system. This is accomplished by supporting the familiar set of file system permissions and ownership data. A file can be associated with a user and a group. Each file then has read, write, and execute permissions for the owner, the group, and all others in the system. These permissions are enforced by the file system service, which requires users to mount the file system using a username and password.

# 4 Implementation Details

The implementation of the filesystem uses the JINI platform. JINI is a Java technology platform for the construction of distributed systems. It was originally developed by Sun Microsystems, Inc. It has been donated to the Apache project under a free license.

## 4.1 Architecture of the JINI platform

The JINI platform provides several benefits when implementing distributed systems. First and foremost, is the SOA design at the heart of JINI makes implemented SOA systems much simpler. In addition, the JINI platform provides several services that can be leveraged by distributed system implementors, such as: service join and service discovery, resource leasing, secure remote method invocation over SSL, system and user authentication via SSL, code mobility, distributed event notification and several other features. However, even though JINI provides a lot of functionality, it does take some effort to properly utilize these features.

The implementation of a service using the JINI platform requires three classes:

- A Java interface to the service, such as MasterService
- A proxy object that can be distributed to clients of the service, such as MasterServiceProxy
- A class that implements the service interface with the actual functionality, such as MasterServiceImpl

The benefit of defining a service in this way is that it separates the interface from the implementation and also provides a small proxy object that the service can distributed in order for clients to use the service. The only classes a client must know about are the proxy and the interface. This is a great benefits for enterprise distributed systems where one design requirement might be that updates and patches can be made to the services without affecting the clients.

In addition to the separation of interface and implementation, the proxy object can be easily utilized for some smart caching or other performance enhancements. For instance, if there are some operations that can be done on the client and some that must happen on the server, the proxy could perform the operations that can be done on the client. In this fashion, the client really does not need to know anything about the service except for the interface. Whether the proxy actually contacts the server or not is irrelevant to the client.

Since each service must contain these three elements, the following convention has been adopted for class names:

- *Service: The Java interface defining the services operations
- *ServiceProxy: The proxy object used by clients
- *ServiceImpl: The implementation class that the server executes

Also, with this architecture we can make some simplifications in the implementation classes to implement multiple services in one class. This has been done with the BlockStorageServiceImpl which implements both the BlockStorageService and the MetaDataService. This simplification was done so it would be easier to test. Instead of starting up n MetaDataServiceImpls and m BlockStorageServiceImpls, we can just start up n BlockStorageServiceImpls and we will have n of each service being provided. These service implementations could easily be broken up into two implementation classes at a later time with no impact on the client.

## 4.2 MasterService Recovery

The MasterService recovery protocol was initially going to be implemented using JINI's distributed event notification service to detect a crash of the MasterService. The current implementation relies on the FileSystemService detecting the crash upon trying to use any of the services of the MasterService. Therefore, the FileSystemService that detects the failure initiates the recovery protocol. The protocol is as follows:

1. The FileSystemService detects a failure of the MasterService (due to either timeouts or other exceptiosn)
2. The FileSystemService sets itself to the not ready state, so that any other requests for services will throw a NotReadyException (a client would just sleep a bit and try again to handle this exception)
3. The FileSystemService selects the MetaDataService with the lowest Universially Unique Identifier (UUID)
4. The FileSystemService then invokes MetaDataService.startMasterServer(listOfCurrentServersToNotify) with a list of servers to notify when the new MasterService is ready
5. The selected MetaDataService starts a new MasterService in a separate thread and initializes it.
6. The MetaDataService then notifies each of the servers in the notification list by invoking joinNewMasterServer(MasterServiceProxy) on each service
7. The services that received the joinNewMasterServer call each call MasterService.join which registers the service with the MasterServer and causes the MasterServer to request all of its metadata and block information
8. When all services in the list have been notified, the new MasterServiceProxy is returned to the FileSystemService
9. The FileSystemService replaces its own MasterServiceProxy with the one returned, sets ready back to true and is now ready for any other requests

Any client that receives a NotReadyException should sleep for a bit of time, and then retry the operation. The freezing of filesystem operations during MasterService recovery is so there are no inconsistencies introduced in the interim period.

## 4.3 Implementation Details by Example

The following example assumes that the client is opening a file for read. The following operations occur:

1. The client calls FileSystemService.open(filename, mode) where mode is read, write, or read/write
2. The FileSystemService calls MasterService.open(clientId, filename, mode)
3. The MasterService traverses the filesystem metadata searching for the filename
4. If the file is found, the MasterService checks the permissions, owner and group of the file against the username and groups of the reqesting client
5. If the access permissions are sufficient, the MasterService checks for any existing locks

(a) A file can be opened for one reader, multiple readers, one writer or one reader/writer

6. If a suitable lock can be obtained, the MasterService returns the file handle to the FileSystemService
7. The FileSystemService then requests the metadata for the file from an appropriate MetaDataService
8. The FileSystemService then returns the file handle to the client
9. The client can then use that file handle for subsequent operations

Now the client wishes to read a block from the file:

1. The client calls FileSystemService.read(fileHandle, blockId)
2. The FileSystemService checks that the fileHandle is valid
3. The FileSystemService queries the MasterService for a list of servers that are currently storing the block for fileHandle and blockId by calling MasterService.locateBlock(fileHandle, blockId)
4. The MasterService looks up in its database which servers are storing the block and returns a list.
5. The FileSystemService then attempts to retrieve the block from one of the block servers in the list by calling BlockStorageService.readBlock(fileHandle, blockId)
6. The BlockStorageService looks up in the blockDB.block_info table to find where on the local disk the block is stored.
7. The BlockStorageService reads the block from the local disk
8. The BlockStorageService decrypts the block with its symmetric encryption key, verifies the integrity of the cleartext data by hashing it and comparing the hash with a hash stored in the database, and returns the block
9. If the BlockStorageService call failed for any reason, the FileSystemService attempts to contact any other block servers in the list to fulfill the request
10. When the FileSystemService receives the requested block without error, it forwards it back to the client.

## 4.4 Deployment

Deployment of the system was made to be very easy and to require minimal external dependencies of the server that is hosting a service. The following are the only requirements imposed on a server in order to deploy a service:

- Java 1.4 or greater must be installed
- jsk-policy.jar must be copied to the Java JRE's lib/ext/ directory
- If using a NAT router or any other firewall, open or forward ports that you would like services to listen on
- Unzip the tarball in any directory with sufficient permissions

- Execute ./scripts/jeri-server.sh (nonsecure) or ./scripts/ssl-server.sh (secure)

The tarball includes the required jars to run the service, including the derby database jar. If the necessary databases do not exist, the will be created during initialization in the current directory.

With this deployment strategy, a single piece of server hardware could easily be used to server multiple services by placing each in its own directory. This is very useful for testing as any number of services could all run on one server.

### 4.5 Issues Encountered and Addressed

There were several issues encountered during the development of this base version of FSFS. Learning how to use JINI was an issue that was overcome through by using examples and the documentation available on the internet. Also, the JINI mailing list was very helpful in answering any questions.

Service discovery is an issue that was solve by using a service provided by JINI called Reggie. Reggie is the default service discovery registrar that comes with JINI. Service discovery can be accomplished via multicast on a LAN or via unicast over a WAN.

Security was an issue that took some effort in configuring. JINI provides the capability to perform remote method invocation over SSL. However, the configuration of this is non-trivial. The current base version has tranport layer security using SSL, however the configuration of ServerAuthentication, ClientAuthentication, and Integrity for the SSL connection are not configured properly in the current implementation. The transport layer is indeed encrypting the data during transport, however the authentication that we wanted to implement is not currently available. Also, there is a currently known problem with the MasterService recovery protocol when using the secure configuration. In order to use the MasterService recovery protocol in the current version, it is suggested that the jeri-server.sh scripts be used instead of the ssl-server.sh scripts. This will disable SSL at the transport layer and use standard TCP connections.

Also, in any production deployment of this system, careful thought would have to be given to solving public key distribution. Currently, each service trusts the services that it utilizes by placing the target service's certificate in its own truststore. Also, currently there is only one public/private key pair per service, so each copy of a service is using the same keys, which would certainly need to be changed for a production release. Also, the passphrase to decrypt a private key is specified in configuration files for ease of testing purposes. Again, in a production deployment, it should be configured to prompt for the passphrase on service initialization.

## 5 Results

In this section we have evaluated our proposed approach in a real world system, comprising of X servers distributed over the Internet. All readings are averaged over $10$ runs and measurements are taken after the system has reached a stable state.

### 5.1 Simulation setup

For measuring the latency in completing the read and write operations, we have assumed a constant block size of $64, \mathrm{Kbytes}$. There are three block-servers in the network that store the file blocks, apart from the essential services like the master-service, the Master service, the Lookup service, File-system service, etc. We use a replication degree of $2$ as it is sufficient for demonstrative purposes. The location and specifications of the

Table 2: Server specifications used by the experimental setup for evaluating FSFS

| Server Name | Location and Specification |
|---|---|
| Server X | Local Test Server: Virtual Machine of Ubuntu Linux |
| Server Y | Local Test Server: Virtual Machine of Ubuntu Linux |
| Server Z | Local Test Server: Virtual Machine of Ubuntu Linux |

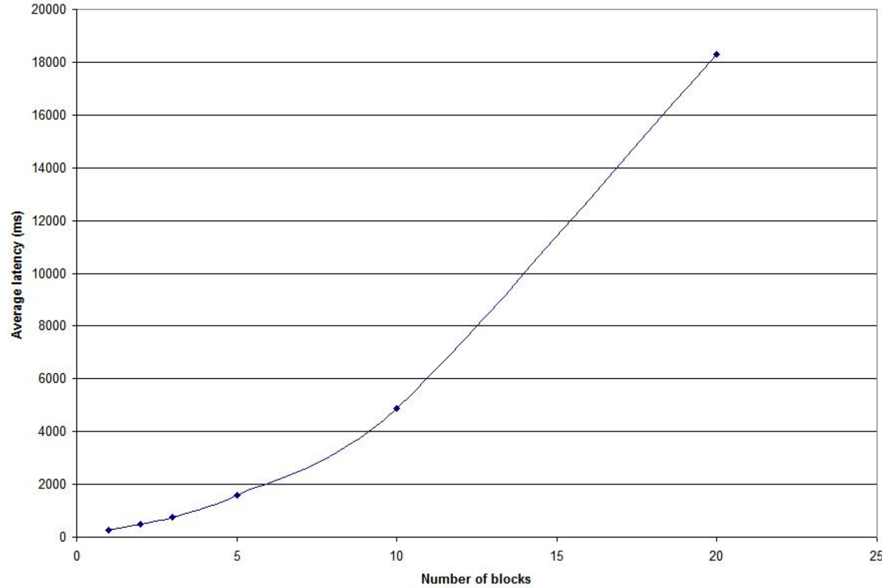


Figure 2: The average latency in milliseconds associated with a file-read operation with increasing number of blocks.

servers is given in table 2. All three BlockStorageServices and MetaDataServices were served off of the same virtual server.

## 5.2 Observations and discussion

Figures 2 and 3 show the graceful performance degradation with increasing loads. For both read and write operations, there the average latency to perform the operation increases. The linear nature of the graph indicates that our systems scales well with large file sizes. Our simulation considers a range of file sizes, from $64, \mathrm{Kbytes}$ to $1280, \mathrm{Kbytes}$ showing good performance of the FSFS architecture.

The latency experienced is a bit high for typical filesystems. There are several reasons for this. First, our test setup used multiple services running from one machine, which was a virtual machine. This was done due to a lack of machines that we had proper access to. Secondly, the current implementation does not implement any caching of data. With the system architecture, strategic caching could be performed at many levels including, but not limited to: the FileSystemService and any of the proxy objects. Lastly, there are several other performance enhancements we could implement in future work that would reduce this latency.

The results, even though a bit high, are reasonable for a base implementation of this type of system design, considering that this file system will operate across a WAN with fully encrypted transport tunnels and also using encryption of the blocks on each data server so that no tampering to the data can be done by a third party after

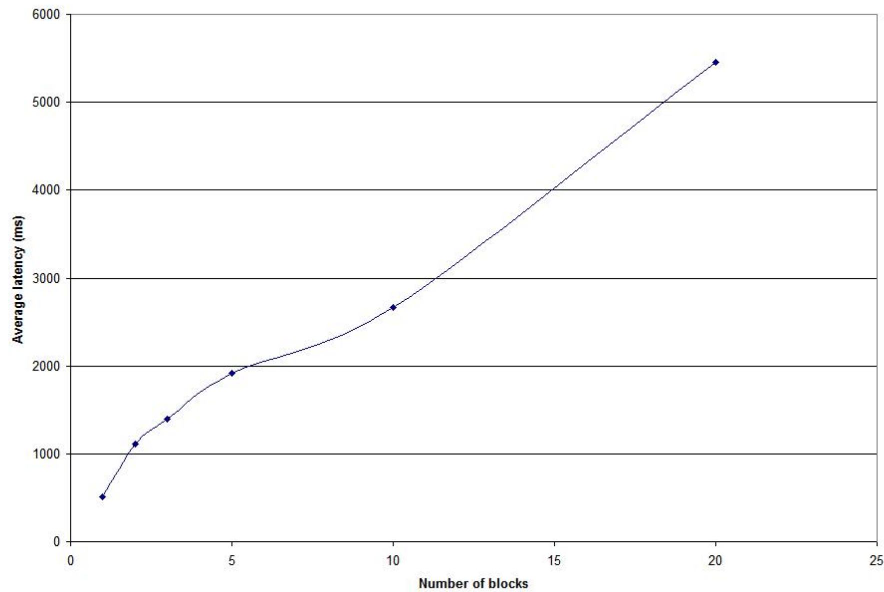


Figure 3: The average latency in milliseconds associated with a file-write operation with increasing number of blocks.

the blocks are stored.

# 6 Conclusion

We have proposed a service-oriented architecture of a distributed file system that is robust to crash failures. Our approach, FSFS, is shown to scale gracefully with increasing file size. Complete reconstruction of the file-system is possible and replication allows for seamless working of the system in presence of multiple server failures. Java allows for platform independence thus allowing easy deployment of our file system on any platform.

# Acknowledgements

The authors would like to thank Dr. Blough for his suggestions and pointers that resulted in the project being in its current form.

The authors would also like to thank the JavaPosse podcast and Van Simmons for the great discussions about the JINI technology platform.

# Disclaimer

Java, JINI and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. The software code and documentation is released on an ”AS IS” basis and the authors claim no responsibility for any incidental or consequential damage to the system on which it is run.

## References


[1] Microsoft Technet, "Website: http://technet2.microsoft.com/WindowsServer/en/library/81cc8a8a-bd32-4786-a849-03245d68d8e41033.mspx?mfr=true."

[2] Mathew Wilcox, "Website: Documentation/filesystems/ext2.txt," in *Linux kernel source documentation.*

[3] R. Hasan, Z. Anwar, W. Yurcik, L. Brumbaugh, and R. Campbell, "A survey of peer-to-peer storage techniques for distributed file systems," in *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II*, 2005.

[4] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere, "Coda: A highly available file system for a distributed workstation environment," *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 447–459, 1990.

[5] Microsoft Technet, "Website: http://technet2.microsoft.com/WindowsServer/en/library/b3754814-f865-4200-9ae9-66785e5e87c81033.mspx?mfr=true."

[6] Napster Inc., "Website: www.napster.com."

[7] The Gnutella Protocol Specification, "Website: http://dss.clip2.com/GnutellaProtocol04.pdf," 2000.

[8] Kazaa, "Website: www.kazaa.com."

[9] Morpheus, "Website: www.morpheus.com."

[10] Ian Clarke and Oskar Sandberg and Brandon Wiley and Theodore W. Hong, "Freenet: A Distributed Anonymous Information Storage and Retrieval System," in *Workshop on Design Issues in Anonymity and Unobservability*, 2000, pp. 46–66.

[11] Jini, "Website: www.jini.org."