DooFuS: A Small, Peer-to-Peer Shared File System

Jack Hood, Eli Meckler, Marc Talbott, Ryan Patton
Williams College
jbh2@williams.edu, ecm3@williams.edu, mbt1@williams.edu, rjp2@williams.edu

Abstract

We describe DooFuS, a peer-to-peer distributed file system intended for small networks of users with limited space who want control over probabilistic data availability at the file level. Distributed file systems render transparent a network of nodes collaborating to maintain data consistency and redundancy to the user who interacts with it as one would a local file system. Peer-to-peer networks use complete functional homogeneity to create highly resilient networks with little established infrastructure. We describe the underlying modules and logic that are responsible for DooFuS' functionality and analyze the application's performance.

1 Introduction and Background

File systems provide an interface for the user to work with memory at the file level. They remove the necessity of managing data consistency, raw memory addresses, or disk failures. They generally provide utilities for viewing metadata about files, such as when they were created, by whom, and how large they are.

A distributed file system exposes the functionality of a traditional file system while abstracting away the underlying distributed nature of the disk storage and all complications that come with it. This is an example of the canonical *transparency* achieved by well-designed distributed systems.

Peer-to-peer networks function with complete functional homogeneity amongst nodes: any node can do all tasks of any other node. Functional homogeneity makes for a very resilient network: since there is no hierarchy, no nodes have raised importance or become points of failure. Peer-to-peer networks have been used to much success in many distributed platforms that require a high level of failure-resistance even when lacking the infrastructure for traditional fail-safe mechanisms.

DooFuS is a peer-to-peer distributed file system designed for a specific use case: small networks of users who want to store and share files of customizable probabilistic availability. This is useful because it lets small groups or individuals securely and easily gain fault tolerance against file loss using just their own computers (and so without forfeiting any privacy). Section 2 discusses the system design and implementation. Section 3 discusses the evaluation of the system. Section 4 reviews related works and future work for DooFuS.

2 Design Overview

2.1 Network

As DooFuS was designed with small groups in mind, its network structure differs from the majority of peer-to-peer systems. The network in DooFuS is fully connected, with every node connected to every other node. In most peer-to-peer networks, a single node will only know about a couple of other nodes (as it would be unreasonable to store information about potentially millions of nodes), which means messages have to spend their time propagating through the network towards their

target. In DooFuS however, the fully connected network lets the system be as fast as possible. Every request or message can be directly sent to its destination. The trade-off is that the system makes heavy use of the network, but since it is intended for small groups of users this shouldn't cause any network overload.

This fully connected nature also means that new information immediately reaches the entirety of the online network, as information updates can be broadcast to every node at once. This removes much of the difficulty and overhead of having to deal with the notion of 'eventual consistency'. Since a node is synced upon joining a network, an entire online network should always be internally consistent (whether or not that consistency is correct is another matter, but not one that is the network's problem). This speed also comes into play with the heartbeats that nodes periodically send out to all other nodes, as it means that news of a node going offline gets to everyone at about the same time. Once again, this comes at the price of heavy network usage, but we find that to be an acceptable trade-off for speed.

A user's identity on the network is determined by an 'id', which can be any string. To connect to a DooFuS network, users reach out to a host that they believe is on the network and send them their id. To be able to connect, or get any type of response, that id must have been manually verified beforehand by someone on the network. So without valid identification, there is no way to connect to a DooFuS node (or even verify that it is one).

In summary, the network design for DooFuS is fully connected and resource intensive. This allows us to achieve the high performance we think is most important for such a small group shared file system.

2.2 File System

The file system is broken into two main components on each node. The file system manager synchronizes with other nodes' managers through the network module and responds to user-triggered commands. The file system backend handles the actual representation of the file system metadata and writing this information to disk.

2.2.1 File System Backend

In order to keep track of replicas of all files, metadata is collected on uploaded files and stored as a json file:

- The filename identifies the file
- The uploader's ID is stored for ownership (associated with possible functionality extensions)
- A list of replicas stores which hosts (by ID) hold copies of the file

All file json objects are stored in one list in a json file. This file is accessed by the File System Backend object. This object maintains a copy of this json structure in soft state. It exposes this object for file addition, file deletion, and replica addition. It also directly exposes deep copies of the underlying structure so the File System Manager can perform more complex tasks that require knowledge of the file list or a file's replica list. The File System Backend exposes deep copies as opposed to references to the objects due to concurrency issues: files may be added or deleted in between calls that depend on length or membership checks.

All File System Backend methods that edit the underlying json object share a common lock that are acquired at the start of the method and released before returning control. This is to prevent simultaneous editing of the json object which could result in undefined behavior.

2.2.2 File System Manager

The Backend is not exposed to other application components by design, which instead work through the File System Manager. The API exposed by the manager is restricted to application-relevant methods.

Along with this restriction, the File System Manager provides the higher functionality of the

file system. This includes local management of file upload and file download, acknowledging new replicas, displaying files to the user, and synchronizing data with newcomer's file systems. As part of its uploading duties, it handles the replication logic (see later section for how replication logic is performed).

The Manager maintains a reference to the network object which is how it initiates many file system utilities that require the cooperation of other nodes.

2.3 Replication Logic

DooFuS is designed so that the system will automatically assign priorities to files (based on how users have marked it). This priority correlates with how likely it will be that the file will be accessible to any user who may want to download it. The higher priority a file has, the more storage it will necessitate on DooFuS since more replicas will be necessary.

To more formally describe the priority-replica relationship: a general availability function A informs us of the probability of accessing a file stored on R replicas out of N nodes total on the network. A value $0 < p < 1$ for an uploaded file is specified such that DooFuS enforces

$$A(R, N) > p$$

by changing how many replicas R must be stored on the network.

To decide upon an R value, we must determine the A function. The availability function A is equivalent to $1 - F$, where $F(R, N)$ is the likelihood that a file is unavailable. F is related to our *failure case*, which is the expected failure conditions of our network. A failure case could be defined in a number of ways. For example, we may say that a given node is a unavailable with probability x, a model used in multiple server systems [1]. In this case, F does not even depend on N:

$$F(R) = x^R$$

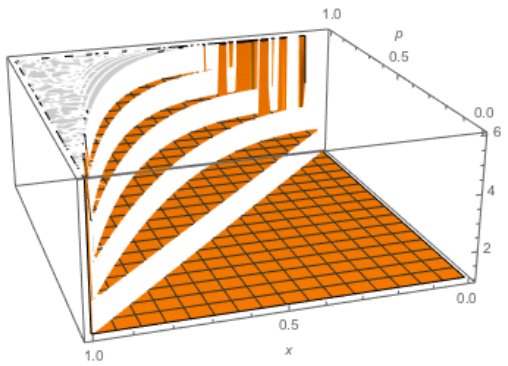

Figure 1: Number of replicas necessary to assure availability of p when any given node is available with probability $1 - x$. Note that the plane in the foreground of the graph is at $R = 1$.

$$p > A(R) = 1 - F(R) = 1 - x^R$$

Solving for R gives us:

$$R = \left\lceil \frac{ln(1-p)}{ln(x)} \right\rceil$$

Another failure case may be that, at any given time, a random fraction x of all nodes N are in a state of failure. This is more complicated, as F (and thus A) is dependent on the total node count:

$$F(R, N) = \frac{\binom{N-R}{xN-R}}{\binom{N}{xN}}$$

$$p > A(R, N) = 1 - F(R, N) = 1 - \frac{\binom{N-R}{xN-R}}{\binom{N}{xN}}$$

Solving for R in this case is non-trivial. Our implementation handles the simpler failure case.

The file system manager, when uploading a file, calculates R using the above formula and then chooses enough nodes so that the network contains the requisite number of replicas to satisfy the priority specified by the uploader.

Currently, file priority is known, but not communicated to other nodes. Since the failure case does not depend on node count, only the initial replica count is important. More replicas do not need to be added as node count increases, so there is no reason to track the priority as time goes on, but

this is a possible extension to be added as outlined in Future Work.

2.4 Uploading a File

The file system manager handles file uploads. There are four steps to the upload process: **file input**, **replica propagation**, **replica storage** and **file and replication data broadcast**.

File input and replica propagation
File input is handled by the backend file writer. The file data is then propagated to nodes on the network according to our replication logic (Sect. 2.3). Each replica node is sent a message containing file data and metadata.

Replica storage
The file system manager of each replica adds file metadata to the file system backend and the backend file writer writes file data to disk. Replicas are treated as dictionary objects and serialized to json format on disk. Replicas are stored this way in order to support file slicing functionality; each dictionary entry is indexed by the part of the file that the stored data reflects. The file writer backend keeps replica data in memory as well as on disk in order to reduce costly reads and writes to disk, and reads each replica into memory after the program is quit and reopened.

The backend file writer holds a lock that is passed to all methods that edit replica json objects. This is to prevent accessing these methods at the same time which could result in undefined modification of the json files.

Data broadcast
After successfully storing file data and metadata, each replica node propagates metadata to the network containing their status as a replica of the file. If nodes don't know about the file yet, their file system manager adds the file and replica metadata to the file system backend. If file upload information has already been propagated to a node, then the node's file system backend simply adds the new replica to its list.

File metadata is not propagated by the uploader unless the uploader is a replica. Instead, it is propagated only by nodes that have successfully stored a replica of the file. In this way, the file is not known by the network until at least one node has written a replica to disk.

2.5 Downloading a File

Downloading a file is also handled by a node's file system manager. There are four steps to the download process: **download request propagation**, **file input**, **file propagation** and **file storage**.

Download requests
When a user requests a file download, they must specify both a file that has online replicas and a directory to add the file to. If the users node is not storing part of the requested file, the backend file writer adds the file to its dictionary and stores the requested download destination. The file system manager then iterates through its list of replicas for the file and chooses which replicas to request the file from. If the file is replicated on the user's node, the backend file writer writes the part of the file it has. The file system manager then sends a request to enough other replica nodes to ensure that it receives all parts of the file. Currently all files are replicated fully on each nodes, so it chooses one replica to request the file from.

File input and propagation
Each replica that receives a file request tells its backend file writer to read in the section of the file that is requested. While a node is online, this data is stored in memory, which makes reading the data much faster than disk storage would. The node then propagates this data, along with metadata containing the part of the file being sent, to the node that sent the request.

File storage
The node that requested a file download receives all file data and tells its backend file writer to write the data to disk. The file writer stores this data as part of the file's contents temporarily in memory without writing it to disk as replica data. Once the file writer has received requests from the manager

to write all parts of the file to disk and has all file data stored in its content dictionary, it appends each file part to the file in order. It write the file to the destination specified by the user.

2.6 Removing a File

When a user decides to delete a file, the file is first removed from the user's local file system metadata. Then DooFuS checks to see if the local node is serving as a replica for the file, and deletes the file itself if it is a replica. Then DooFuS broadcasts a message to the rest of the network telling the nodes that the file is being deleted from the file system. Each node performs the same process as the original node that issued the delete file request. The node first removes the file from its file system metadata before checking to see if it was serving as a replica for the file and deleting the file itself if so. Once this process is completed, the file has been deleted from the file system for all of the nodes currently connected to the network.

There is one downside to this approach. If nodes that know about the file are not currently connected to the network when the file is deleted, those nodes will not receive the delete file request and will not delete the file as a result. When the node rejoins the network, it's file system metadata will propagate throughout the rest of the network. This will essentially re-add the file to the distributed file system and undo the original delete file request. This presents an interesting challenge, but is not one we chose to address in the scope of our project. We are assuming that our network is small and fully-connected. Moreover, we are assuming that all nodes are constantly online. As a result, we are not going to address this issue and will instead assume that all nodes that know about a given file are connected to the network when a delete file request for that file is issued.

This challenge could have been addressed in a number of ways. The original node that issued the delete request could have continued trying to delete the file until it was certain that every node that had once known about the file had been told to delete the file. This approach would have been extremely resource intensive, and it is possible that the original node would wait indefinitely for a node to connect to the network. Another, more reasonable approach to this problem, would be to implement a blockchain. The blockchain would serve as a distributed record of all file deletions (and possibly additions if we chose). As we learned during some of the final presentations in class, a blockchain is a means of storing a distributed record that can be used to maintain consistency in peer-to-peer networks with regularly entering and exiting nodes. By using a blockchain to store all delete requests, we could ensure that nodes entering the network that had not been informed of a deleted file would find out immediately once they joined. This would prevent the situation that we described earlier when a new node re-adds a deleted file to our file system. While we did not add this blockchain functionality, it would be a relatively simple addition in the future.

3 Evaluation

In order to evaluate the performance of DooFuS, we ran a couple of tests. The main metric that we were concerned about was the quickness of the system. Users always expect speed from their file systems. They expect to be able to create new files, find them, and delete them rapidly. These expectations do not change when using a distributed file system. In the interest of transparency, the distributed nature of a distributed file system should be hidden from the user. Therefore, the file system should operate quickly, whether it is distributed or not. DooFuS is not exempt from these user expectations. As a result, we also needed to ensure that DooFuS ran at a speed that was acceptable for use as a file system.

3.1 Speed

When testing speed, we determined that the two most important features of DooFuS that needed to execute quickly were uploading and downloading of files. We chose upload speed because this is how users will add newly created files to the file system. If users have to wait for long periods of time whenever a new file is created, they will not want to use

Replicas	Upload Speed
2	0.040238
3	0.032153
4	0.030265
5	0.036480

Table 1: Upload Speeds

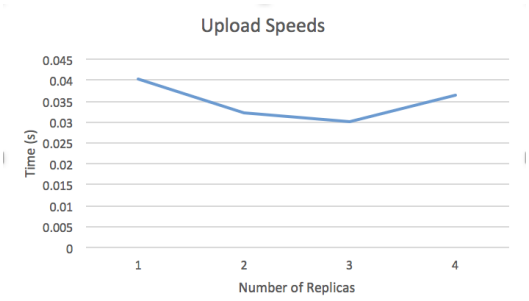

Figure 2: Upload times for varying levels of nodes

DooFuS. Creating files is a common task in file systems, so slow execution here will make the entire system appear slow. Downloading files is important for similar reasons. Users will download files when they want to edit or otherwise access the files or when they want to access files that have been shared with them by their friends on the network. In either of these cases, the user will want to be able to open these files quickly. Ensuring rapid download speeds will allow them to do so and prevent DooFuS from appearing slow.

3.1.1 Uploads

We tested how long it took for a file to be uploaded and replicated across our network with varying numbers of replicas. We anticipated the upload time increasing as we required more replicas because the file would need to be transferred to more nodes. However, as Table 1 and Figure 2 illustrate, this was not the case. This is likely because the actual time taken to transfer the file over the Internet eclipsed the time taken by our system. As a result, this would obscure the running time of our system as we replicate a file over increasingly many nodes. Nonetheless, the upload speeds that our system was able to attain are adequate and will not prevent a user from adopting DooFuS.

Replicas	Download Speed
2	0.013715

Table 2: Download Speeds

3.1.2 Downloads

To test the download speeds of our file, we measured how long it took to download a single file from a network with only two replicas. We did not bother measuring the download times on networks with larger replica counts because our system currently downloads a file from the first replica on its replica list. As a result, adding more replicas would not impact the download speed. The download speed can be seen in Table 2. Clearly, this speed is also acceptable. A download that only takes this long will not be noticed by a user. As a result, the user will find our system to be adequately fast for downloading files. When considered alongside our upload speeds, we have found that DooFuS functions fast and has the potential to serve as a user's primary file system.

3.2 Consistency

Deciding how to achieve consistency is a key part of any distributed system, and one we thought about a lot. For our purposes, we decided that consistency has two parts: One is that once a file is added to the network it will never be unintentionally lost or have incorrect metadata associated with it (as to who is holding what replicas). We were able to ensure this property, but have yet to implement the second part which is that a deleted file stays deleted. This is a critical flaw as it means that once a single user is offline during a deletion the system has lost consistency. However we believe there to be a fairly simple fix, and once we have deletion consistency our system should be functionally consistent.

4 Discussion

4.1 Related Work

DooFuS is not the first application to store data on a peer-to-peer network. Chord is a distributed lookup protocol designed to help efficiently locate data on a peer-to-peer network. Chord can be used

to build peer-to-peer distributed hash tables which would allow a user to store data on a peer-to-peer network and access that data quickly and efficiently. Chord seeks to provide efficient lookup of data, but since we focused on a small, fully connected network, this was not necessary. DooFuS is able to locate data far more efficiently than Chord because it knows where all of the data is stored on the network. If we wished to expand our network to the point where it was no longer fully connected, Chord might be a useful tool to ensure that our network still located data quickly.

There are also examples of peer-to-peer distributed file systems. One of these systems is known as the InterPlanetary File System (IPFS). IPFS is a peer-to-peer distributed file system that bills itself as a hypermedia distribution protocol that could replace HTTP. [2] The idea behind IPFS is to connect all computers to a shared distributed file system, which will allow users to access files hosted in the peer-to-peer network. This is a similar goal to our system, albeit with much loftier goals and a larger scope. We do not hope to redesign the Internet, and our goals are designed to reflect that. DooFuS has shown promise as a social network (see our poking functionality for evidence), but it has been designed and implemented with small-group peer-to-peer networking in mind. Nonetheless, IPFS does show the immense potential that peer-to-peer distributed file systems offer. There are also other peer-to-peer distributed file systems that have been developed such as Shark and Black-box, which focus their efforts on different goals such as efficient caching and security. [3] [4]

4.2 Future Work/Possible Extensions

File type support
Currently DooFuS provides functionality for text files. We plan to extend this functionality to include all file types. Our general system structure allows for this change. However, there are currently steps in the upload and download process that require file data to be decoded to string format. One issue is that json objects are stored in string format. To account for this functionality, the file writer must write each part of binary file data separately from the json file, which stores only metadata. Also, the current network message format casts all messages to string format before encoding and sending data across the network. This could be changed to support binary file data by only casting metadata to string format and sending file data without decoding it.

File slicing
The backend file writer and the network were implemented to support dividing files into slices and sending each slice to a different node as part of the replica propagation process. We plan to support file slicing in the future. To implement file slicing, the file system backend must store additional replication metadata about which part of the file is stored on which replica, so that the file system manager knows which replicas to request which file parts from. Once nodes know this information, the rest of the system is already equipped to support slicing functionality.

Encryption and Digital Identities
To increase the security of DooFuS, we would like to add encryption at all levels. We would like to encrypt all data sent across the network, as well as all data stored in local config files. This would prevent MitM or network monitoring attacks from gaining any information about the files or the users on the network (the first is troubling because it violates the privacy goal of the system, and second is troubling as it would give those attackers enough information to actually join the network with its current handshake mechanism). One way to implement this would be with public-private key encryption, where every user (or more specifically, each computer) has a 'digital identity' that is their public key.

5 Conclusion

5.1 Reflection on the Assignment

We encountered a lot of issues implementing a peer-to-peer distributed system that would not have come up if we had used a centralized model. For example, our system must propagate new information to all nodes every time a user changes

the state of the distributed file system. This sets up problems with synchronicity as well as reliability. It is difficult to make sure that all peers are working with the same state.

We also discovered benefits to peer-to-peer systems. Privacy is a huge benefit over a centralized model, because when you upload a file to DooFuS it is split up on the machines of a few trusted users instead of stored on a company's servers. Another benefit is reliability. We don't have a single point of failure. Instead, our replication logic ensures that files are replicated on a number of nodes based on their priority, which means important files are not lost if a node goes down or intentionally leaves the network.

6 Acknowledgements

Thank you to Henry Lane and Molly Knoedler for help working through the replication math and another thank you to Molly for the help with plotting in Mathematica.

References

[1] Sami Rollins. Replication, 2008 (accessed December 14, 2017). http://www.cs.usfca.edu/ srollins/courses/cs682-s08/web/notes/replication.html.

[2] Ipfs - the permanent web, 2017 (accessed December 15, 2017). https://github.com/ipfs/ipfs.

[3] David Mazieres Siddhartha Annapureddy, Michael J. Freedman. Shark, 2005 (Accessed December 15, 2017). http://www.scs.stanford.edu/shark/docs/shark-nsdi05.pdf.

[4] Syed S. Rizvi and Abdul Razaque. Blackbox, 2016 (Accessed December 15, 2017). http://ieeexplore.ieee.org/document/7840163/?part=1.