Byzantine Fault-tolerant RAID-like Filesystem

Adrien Wu
adrienwu@stanford.edu

Brandon Lou
brlou@stanford.edu

Joe Tsai
joemht@stanford.edu

Abstract

We design and implement a distributed, Byzantine fault-tolerant RAID-like filesystem capable of tolerating up to $\lfloor \frac{n-1}{2} \rfloor$ Byzantine faults and a configurable number of crash faults. This enables multiple parties to collaborate on a shared set of files when they do not trust a centralized party or even each other. We combine digital signatures and error-correction codes to prevent malicious parties from performing unauthorized modifications, and to reconstruct data from corrupted sources due to malicious or down servers. Our filesystem implementation integrates with FUSE, allowing existing applications to interact with it using ordinary I/O operations to provide a seamless user experience.

1 Introduction

We designed the Byzantine fault-tolerant RAID-like (BFR) filesystem to enable multiple parties to collaborate on large files without trusting a centralized party or each other. For instance, multiple people may want to work on some large videos together, where one person edits a video and others watch the edited videos. In other words, users perform reads more frequently than writes. They do not trust a centralized third-party, such as Google Drive, to store their data, due to say availability or privacy concerns. These users decide to share the files among themselves, but no single peer wants to store the large files, and peers may not even trust a single peer to store them all. Here is where our distributed filesystem comes into play, allowing these people to store portions of files among themselves, while also preventing any user from modifying files they are not authorized to.

The BFR filesystem consists of a fixed number of servers such that some of them may be malicious or unreachable. Clients operate with these servers via remote procedure calls to perform operations such as creating, reading, writing, and deleting files. When writing to a file, clients encode their data using error-correction codes, so that even if parts of their data are lost or corrupted, their original data can be recovered. Clients also sign their data to ensure it was not tampered with. Clients split the encoded and signed data into blocks and store them on multiple servers. To read a file, clients retrieve parts of their data from multiple servers, verify their signatures, and reconstructs their data using error-correction codes.

Since servers may sometimes crash or become unreachable through the network, they may miss the latest file updates from clients. To retrieve a file's latest version, we implemented a heartbeat mechanism such that servers can detect whether their files' version falls out of date. We implemented a recovery mechanism such that servers can learn which portions of their files have been updated. Knowing this information, servers can then read the latest file data from other servers.

Our implementation uses gRPC for client/server communication and a FUSE wrapper to enable existing applications to use our filesystem out of the box. We evaluated our filesystem by measuring the encode and decode throughput across various block sizes, and the read and write latency across various request sizes.

In the future, we hope to add support for multiple owners of a single file. Currently, each file only supports one owner with write permissions, but multiple users can read. We also hope to improve our filesystem performance by reducing disk I/O through caching.

2 Design

2.1 Assumptions

Our BFR filesystem consists of n servers, indexed from 0 to $n-1$. Of the n servers, users specify the maximum number of servers that can act in a Byzantine manner b, and the maximum number of faulty servers for successful reads f. We only require $2b \leq f < n$, which implies $b \leq \lfloor \frac{n-1}{2} \rfloor$, but an optional $b \leq \lfloor \frac{n-1}{3} \rfloor$ constraint can be added to avoid a minor drawback which will be further detailed in the discussion of retrieving the list of files. We define "faulty" servers as the number of servers that are either unreachable or act in a Byzantine man-

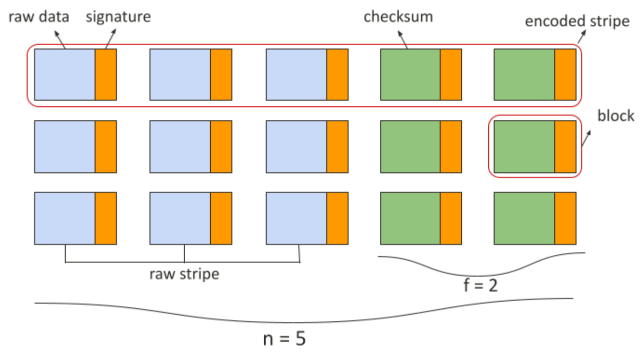

Figure 1: Diagram of blocks and stripes when $n = 5$ and $f = 2$

ner. Each operation has its own limit of faulty servers, which will be detailed in Section 2.6.

2.2 Data Layout

We divide each file into equal-sized **raw stripes**. For files not a multiple of the raw stripe size S, the last stripe is padded until to a multiple of the stripe size, and we record the real file size separately. Each raw stripe can be encoded into an **encode stripe**, which is further divided into n **blocks**, where for all $i \in [0, n-1]$, block i is stored on server i, as shown in Figure 1. The user specifies the block size B, and the raw stripe size is calculated as $S = (B - ns)(n - f)$, where $s = 64$ bytes is the size of one Ed25519 signature.

We divide files up this way in order to use Reed-Solomon error-correction codes [5]. A raw stripe is encoded such that any f of the n blocks in an encoded stripe can be missing, but the entire raw stripe can still be reconstructed. The raw stripe size affects the granularity of data that can be encoded and decoded at a time.

Each block of encoded data is signed along with (filename, version, stripe ID, server ID) by the file owner's private key, and servers store encoded and signed blocks for every file. Anyone with the owner's public key can read and verify that the data was written by the owner. Since only the owner has their private key, malicious servers cannot modify the owner's files. The n signatures of all blocks in an encoded stripe are replicated over all the blocks for recovery purposes (hence the $(B - ns)$ term of the raw stripe size), but only the signature corresponding to the block is needed for verification.

Each client has their own "workspace" under the root directory, which is a directory whose name is the client's Base64-encoded public key. This structure ensures that a file's public key information never changes, and malicious servers cannot fake a file's public key.

Thus, malicious servers can only (1) decide not to reply, (2) ignore some files' existence, (3) reply with signed old versions of files, and (4) add some files on their servers with their own private key(s).

Although we assume some servers to be malicious, we assume all clients performing write operations (e.g. create, write, and delete) to be honest. We did not implement any client checks in our implementation, but future work includes an authentication system to the client-server interaction.

2.3 Versioning

To ensure file readers receive a consistent view of stripes, a version number is tracked for every file. This version is also used in other purposes such as recovery and signature verification. Every create, write and delete operation increments the version number by one.

For each version, we define its **update log entry** as containing (1) the version number, (2) the file size after the update, (3) the range of stripes it updates (each update can only update one contiguous range of stripes), (4) whether it is an create/delete operation, and (5) the signature of the above data. The signature ensures that the entry can be verified in the future. The list of all update log entries of a file will be referred to as its **update log**. The update log contains all necessary information for the servers to determine the current state of the file.

Additionally, servers store the data necessary for recovering the data blocks of any recent versions so that clients can request to read a specific version. This is because during ongoing write operations, some servers may be ahead of other servers, and if we only keep the most recent version, the read operations will stall. The recent version data will be referred to as the **undo log**, since we use a similar mechanism as undo logs used in databases in our implementation. Because the undo log is only used for reconstructing a recent version, the undo log of a version will be deleted after a short time-to-live, which we chose to be 30 seconds in our implementation. It is worth noting that while the undo log will be deleted after a timeout, the update log will be kept until the file is deleted.

2.4 Error correction

In our implementation, we use the Reed-Solomon error correction code $RS(n, n-f)$ on the Galois field $GF(2^8)$ for simplicity. This places a constraint of $n \leq 255$; however this can be solved by using Reed-Solomon codes on a different field with an FFT-based implementation such as [4]. Because we can identify corrupted blocks by verifying their signatures, we used our own Reed-Solomon implementation that only deals with erasure errors and optimizes decoding performance, contrasting most existing implementations that allow error identification and optimize for encoding performance.

2.5 API Overview

All servers expose the following APIs. In the next section, we will describe how these APIs are used in the context of various file operations.

1. `GetFileList([fileName, includeDeleted]) --> files`
2. `WriteBlocks(fileName, blockData, updateLogEntry) --> status`
3. `CreateFile(fileName, updateLogEntry) --> status`
4. `DeleteFile(fileName, updateLogEntry) --> status`
5. `ReadBlocks(fileName, stripeRange, version) --> (blockData, updateLog)`
6. `GetUpdateLog(fileName, afterVersion) --> updateLog`

2.6 Operations

Retrieving list of files

Clients call `GetFileList` on all servers to retrieve the filenames and the most recent update log entry of all files by default. If clients optionally pass in a filename, servers only return the information of that file. Clients set `includeDeleted` to true if they want to obtain information about files that have been deleted. This will be used when creating files.

File creation and the heartbeat mechanism use the `includeDeleted` field. We will describe heartbeats in greater detail in section 2.7. The heartbeat process needs to know whether a file has been updated, and we consider deletion to be an update, so `includeDeleted` is set. For file creation, the client needs to generate the update log entry, so it needs to know the version number. Thus clients will need to set `includeDeleted` to retrieve the current version number to account for the case if a file with the same name has been previously deleted.

A client will wait for $q = \min(2b+1, n-b)$ servers to respond, giving a tolerance of $n-q$ faulty servers. The client first verifies each of the returned files using the owner's public key obtained from the filename and the signature in the update log entry. Then, it counts the occurrences of each file. If a file is returned by at least $q-b$ servers, the client assumes that the file exists; otherwise, the client assumes that the file does not exist.

If $n \geq 3b+1$, we have $q-b \geq b+1$ so all files that the client assumes exists are valid because at least one of them was returned by an honest server. If $2b+1 \leq n < 3b+1$, then $q-b$ colluding malicious servers can add "phantom files" with their own public keys to their response to trick the client into believing those files exist. However, because filenames contains owners' public key, these files will not affect actual files written by clients. Even if clients actually try to read the file, the malicious servers alone cannot make up enough blocks to form a successful read, so these phantom files pose little effect on the client.

Writing to a file

To write to file, the client first calls `GetFileList` to retrieve the version to generate the update log entry. After that, the client calls `WriteBlocks` specifying the filename, the update log entry, and an array of blocks. This array contains one block from each stripe. Clients call `WriteBlocks` on all servers, but with different arrays of blocks. Specifically, server i only receives blocks at position i within each stripe.

The server first checks if the file exists and has not been deleted, and optionally verifies the signed block and the signed message with the owner's public key. After that, the server creates the update log entry and the corresponding undo log for the requested write, and then commits the blocks into its file.

We must ensure at least $n-f$ honest servers contain its portion of data in order for error correction to succeed. We also have to wait for b more servers to account for at most b Byzantine servers falsely replying. Hence, clients must wait to receive at least $n-f+b$ successful responses before considering the write a success, giving a tolerance of $f-b$ faulty servers. This same reasoning applies to the minimum number of responses for creating a file and deleting a file, as they can be thought of as special cases of write.

If the client receives less than $n-f+b$ successful responses, the file is left in an undetermined state. Thus, the client must retry the write request until receiving $n-f+b$ successful responses before making any other updates to the same file.

Creating a File

To create a file, the client first uses `GetFileList` with `includeDeleted` set to true to retrieve the version. The client sets `includeDeleted` because when a file with the same name is created after being deleted, the version starts from the previous one instead of being reset to 0. If the file never previously existed, then the client sets the version to 0. Otherwise, the client increases the version. The client signs the update log entry and call `CreateFile` on all servers.

Servers first verify that the file to be created does not exist or has been marked as deleted. Then, servers adds the new file to their file list, allocates storage for the file, and stores the update log entry. This operation can be seen as a special case of writing a file.

Clients wait for at least $n-f+b$ successful replies before considering the file creation to be successful, for the same reason as described in the write operation.

Deleting a file

To delete a file, a client first calls `GetFileList` for the latest version of the file and generates an update log entry, similar to writing a file. The client then calls `DeleteFile` on all servers, specifying the filename and attaching the update log entry.

A server first checks for the existence of the file and verifies the update log entry. Then the server removes the content of the file and deletes its associated update logs and undo logs. Finally, the server stores the update log entry similar to that in file creation.

Clients consider the file to be deleted if $n - f + b$ servers successfully acknowledged the deletion.

Reading a file

To read a file, a client first calls `GetFileList` to determine the most recent version. After that, it calls `ReadBlocks` on all servers, specifying the filename, the first stripe to read, the number of following stripes to read, and the version that it obtained earlier.

Each server checks that the file has not been deleted and that the version exists. If any of these conditions do not hold, the server responds with a error status. If the requested version equals the latest version, the server loads the blocks in the specified range from disk and returns them to the client with a successful status. If the client specified an older version, the server reconstructs the version from its undo log entries that were generated when the version was updated. By using a binary tree to scan through the update log, the reconstruction runs in $O(\text{ReadSize} + v \log v)$ where v represents the number of versions between the requested version and the current version.

After retrieving the responses, the client then processes the update log to calculate the last modified version for each stripe. This is because the block data of a stripe is signed with its corresponding version number, which will most likely differ from the current version number. Then, the client verifies the signatures on each block to ensure their integrity. Finally, the client decodes the blocks at a stripe level.

The client collects at least $n - f$ encoded blocks for each stripe because the decoding algorithm requires at least $n - f$ correct blocks. The client verifies the signatures on each block to ensure their integrity. If the number of correct blocks is less than $n - f$ (meaning some Byzantine-faulty servers returned corrupted blocks), the client waits for further responses (e.g. the client finds that only $n - f - 4$ blocks are correct so it waits for another 4 responses before trying to decode again) until reaching $n - f$ correct blocks and decoding the raw data. This gives a tolerance of f faulty servers.

2.7 Server Recovery

Servers may sometimes crash or be unreachable due to network issues. In these situations, they may not receive the latest file modifications from clients. To maintain a healthy state, servers perform periodic heartbeats, and perform a recovery process for files detected to be of an older version.

Heartbeat

Servers periodically execute a heartbeat mechanism to retrieve up-to-date file versions because some previous write requests may have failed to reach particular servers. To perform a heartbeat, servers call `GetFileList` on other servers to retrieve the latest version number (target version) of each file. After collecting the replies, a server checks if any of its replies contain a version greater than its current version. If so, the server knows its file is not up-to-date and will start the recovery process for that file.

To reduce the traffic of heartbeats, we applied an optimization in our implementation by sacrificing some recovery time, utilizing the fact that it is acceptable to miss an up-to-date version in an heartbeat provided servers will eventually receive the version. Thus, for each heartbeat operation, each server only queries a subset of the servers. The list of servers queried in a heartbeat is chosen in a round-robin fashion, so after a fixed amount of time each server is queried at least once.

As previously discussed, malicious servers can only fake phantom files with their own private keys; thus files with an already-seen public key are guaranteed to not be a phantom file. For each file with unseen private keys, servers will keep track of how many other servers returned that file. Once this exceeds $b + 1$, a server knows the particular file is not a phantom file and can start the recovery process if needed.

Recovery

The naive approach to recovering a file is to read the entire updated file from other servers. However, we can optimize this process since the update log can be used to calculate which stripes have been modified from the current version to the target version. Hence, to recover the latest version of a file, a server calls `GetUpdateLog` from all servers and waits for $b + 1$ responses. Then the server calls `ReadBlocks` on the modified parts of the file, similar to what a client would do when reading a file. Finally, the server reconstructs its own portion of the blocks and updates its own update log. Thus, recovery requires $n - f$ honest servers other than the recovering server itself.

Recovery is also the reason why signatures are replicated across all servers. Otherwise, servers would not be able to reconstruct the original block because they do not have the private key of the owner.

3 Implementation

We implemented our filesystem in around 4000 lines of C++. We used gRPC for communication between clients and servers. Clients and servers use a common library that we implemented to encode/decode and sign/verify stripes. We implemented a Filesystem in Userspace (FUSE) wrapper around our client filesystem, enabling end-users to use UNIX system calls (e.g. open, read, write, close) to interact with our filesystem.

3.1 Server

Servers run on top of the machine's underlying filesystem (e.g. NTFS, APFS, ext4). When a server receives a write request, it generates the undo log for the current version and stores it along with the update log entry in a dedicated directory as a binary file. After successfully persisting the undo log, the server can safely update the current file. The server concatenates all the blocks and writes it as a binary file to its filesystem.

To serve a read request, a server translates the starting stripe offset and stripe count into a byte offset and byte size relative to the local file. Then the server reads the file data from its local storage.

When a server recovers from a crash, it scans its local storage for any existing files and then loads its undo log and update log to resume from its previous state.

3.2 Client

FUSE operation	RPC
`create`, `open`	`CreateFile`, `DeleteFile` (`O_TRUNC`)
`unlink`	`DeleteFile`
`read`	`ReadBlocks`
`write`	`ReadBlocks`, `WriteBlocks`
`getattr`, `readdir`	`GetFileList`

Table 1: Main RPCs used in each FUSE operation

FUSE provides a set of operations we can implement to enable end-users to continue using UNIX-like filesystem operations. This enables existing programs to work out of the box on top of our filesystem. Specifically, we implemented the following FUSE operations: `open`, `getattr`, `read`, `readdir`, `write`, `create`, `unlink`, `mkdir` and `rmdir`. These operations were sufficient to run simple programs such as `ls`, `cat`, `echo`, and `vim`.

For most FUSE operations, their implementation was trivial by calling the corresponding RPC functions. We list the table of operations in Table 1.

We implement `open` with an `O_TRUNC` flag by a file deletion followed by a file creation. Note that for `create`, `write`, and `unlink` operations, clients also need to call `GetFileList` to obtain the current version number in order to generate the update log entry, as discussed in Section 2.6. Additionally, if a `write` does not align with the raw stripe boundary, the client needs to first read the first and/or last stripe so that the file can be properly updated.

We also implement `mkdir` and `rmdir` by locally storing the list of empty sub-directories (inside a client's workspace). The server's implementation does not include concept of directories (other than the first-layer public key directory), but it allows arbitrary filenames including those containing slashes (`'/'`). Thus, FUSE clients must parse the list of files to get the list of subdirectories when performing `getattr` and `readdir`. This means any empty sub-directories will be lost once the FUSE client restarts, but sub-directories containing files will be preserved.

Since all of our operations involve communicating with multiple servers, such as retrieving data from multiple sources, and writing blocks to different servers, we used asynchronous I/O provided by gRPC so that the client can send multiple requests to multiple servers at the same time.

4 Evaluation

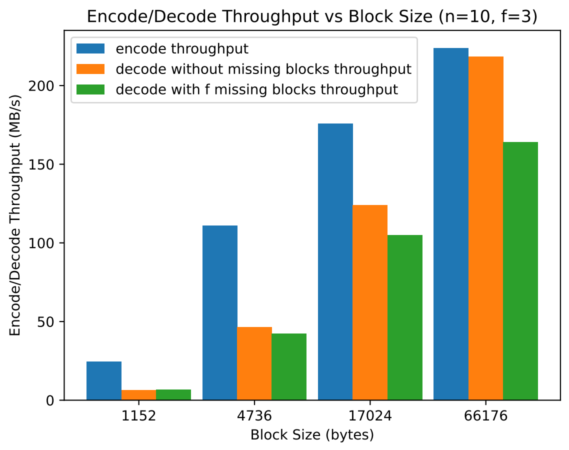

Figure 2: Encode/decode throughput over various block sizes

We measured the encoding and decoding throughput to determine the most suitable block size for our filesystem. We ran an experiment with $n = 10$ servers and up to $f = 3$ failures. We used a Google Cloud Platform e2-highcpu-16 machine with 16 CPU cores and 16 GB of memory. However we only used 4 of the cores for parallel encoding and decoding. The results in Figure 2 show that with a larger block size, the overhead of the signature size decreases and thus throughput increases. We noticed that when using a block size of $17024 = 2^{14} + 64n$ bytes, both encoding and decoding reached 100 MB/s throughput. We felt this was sufficiently fast in a filesystem so that encoding and decoding would not be a bottleneck, and the resulting raw stripe size of 160 KiB was also reasonable.

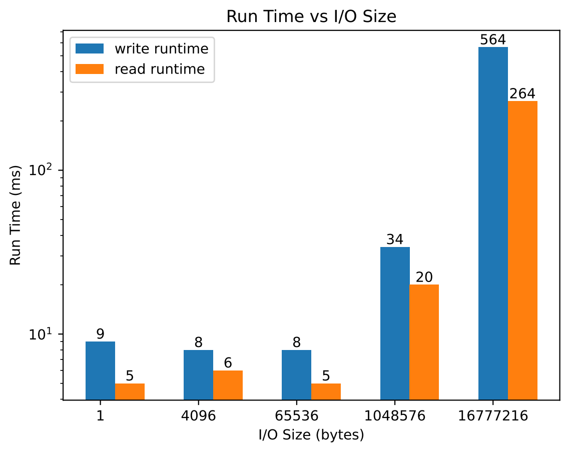

Figure 3: Read and write latency over various request sizes

In our next experiment, we measured the runtime of reads/writes of different sizes with parameters $(n, f, b) = (10, 3, 0)$ and a block size of 17024 bytes with f servers down. The results in Figure 3 shows a read latency of 5 ms and write latency of 9 ms. We achieve read throughput of 63.5 MB/s and write throughput of 29.7 MB/s. The throughput is mainly limited by the write throughput of server disks. We also repeat the same experiment with all n servers online and the results were the same (within error margins). This is expected because the read operation only waits for $n - f$ responses.

5 Related Work

Our BFR filesystem partially took inspiration from Houtman and Ousterhout's work on the Zebra striped file system [2]. This formed the basis of our RAID-like striping mechanism. However, Zebra focused on optimizing small write-heavy loads (aggregates small write requests together), and only tolerates one down server, while our filesystem tolerates Byzantine faults and a configurable number of down servers.

Distributed filesystems like ours can also be thought of as state machine replication. Many works have tackled the problem of Byzantine fault-tolerant SMR, such as Castro and Liskov's PBFT algorithm [1]. The authors applied PBFT to the Network File System protocol.

Our motivation also shares similarities with Li et al.'s SUNDR filesystem [3] and the mechanisms have some resemblance. For instance, both BFR and SUNDR do not present the application incorrect data even if servers are compromised. In BFR, if a malicious server provides incorrect blocks, the client's block signature verification would fail.

6 Future Work

In our current design, we only allow a single client to modify the file, which we believe to not always be an ideal constraint. However, allowing multiple clients to modify the same file is challenging. This feature involves handling version conflicts and maintaining signatures of different authors. This requires incorporating a consensus mechanism and would require significant changes to our design.

We also hope to further improve the performance of our filesystem. In our current implementation, servers must perform disk I/O on every read and write request. To serve `ReadBlocks`, a server loads the content of the file from disk; to serve `WriteBlocks`, a server writes to disk the undo log and the new data. One improvement would be to introduce a caching mechanism such that we keep as much data in memory as possible. Reducing disk I/O and operating with memory could lead to huge performance gains. The challenges for caches include maintaining consistency and designing a performant cache eviction policy.

7 Conclusion

Our BFR filesystem solves the problem of multiple parties collaborating on large shared files without trusting a centralized party or each other. Corrupted files can be recovered using error-correction codes and unauthorized modifications can be caught using digital signatures. We designed a set of RPC calls for clients to interact with files on servers, and introduced a recovery mechanism for servers to catch up with latest versions of files. We implemented a FUSE wrapper around our client, allowing out-of-the-box integration with existing applications for a seamless user experience.

References

[1] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, page 173–186, 1999. https://pmg.csail.mit.edu/papers/osdi99.pdf.

[2] John H. Hartman and John K. Ousterhout. The zebra striped network file system. In *ACM Transactions on Computer Systems*, page 274–310, 1995. https://doi.org/10.1145/210126.210131.

[3] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure Untrusted Data Repository (SUNDR). In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume*

6, pages 9–9, 2004. http://dl.acm.org/citation.cfm?id=1251254.1251263.

[4] Sian-Jheng Lin and Wei-Ho Chung. Ieee transactions on information forensics and security an efficient (n, k) information dispersal algorithm based on fermat number transforms. https://api.semanticscholar.org/CorpusID:14469905.

[5] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960. https://doi.org/10.1137/0108018.