PP2: A Raft-Based Distributed File System

Jay Lang, Timmy Xiao, Brandon Yue
https://github.com/jaytlang/pp2

Abstract
We present Pigeon Protocol 2 (PP2 for short, a spiritual successor to our group's 6.033 system design project), a distributed file system built with consistency and availability in mind. Our design leverages a pre-existing Raft implementation designed for 6.824 Labs 2-3 to simplify file replication and offer strong consistency in the face of non-Byzantine errors. The filesystem is journaling, offering atomic writes and a POSIX-like API. We evaluate our filesystem for its correctness and demonstrate its capabilities.

Blocks
We use a Raft Key/Value server to provide a fault-tolerant, strongly consistent way to store blocks: block numbers translate to keys, and values represent block contents (strings) capped by the 4096-byte block size. Blocks are foundational units for inodes and everything else in the filesystem. Access to blocks is atomic, as the underlying atomic Raft operations (Get/Put/Append with minimal modifications) operate directly on the key/values that constitute these blocks.

Block Cache and Concurrency
Using a Raft implementation as an underlying block device allows us to construct strong concurrency primitives replicated across many servers. We maintain a set of lock blocks, which are one-to-one mapped with data blocks in our distributed block device. Assuming that all clients run on a roughly synchronous clock, clients may lock a block by attempting to write a 1-second-resolution timestamp to the corresponding lock block.

This test-and-set primitive is atomic, as it is a custom operation within each Raft server. Therefore, a double-acquire is impossible. A client may release a lock block by sending an empty block to the Raft cluster. Otherwise, the lock will expire after five seconds per the above algorithm, and another client may claim it. When this occurs, servers inform the previous holder of the exchange in ownership. All servers replicate the new client's timestamp, and the old client can no longer issue Get/Put/Append operations until lock reacquisition.

This locking system enables us to construct a simple block cache, with an enforced invariant that only one client may hold each block at a time. This feature occurs in most non-distributed file systems, moderates access to each block to prevent data races, and enables us to build higher-level constructs cleanly.

Journaling

The filesystem features a data-mode journal permitting fully atomic writes of up to 100 blocks at a time. During a system call, client code may elect to begin transactions in which writes are atomic. Up to 1,000 transactions may occur at a time. The journal persists the number of outstanding transactions, and when a client notices this count drop to zero, it gains an exclusive lock on the log, commits it, flushes new blocks to the block device, and then resets the journal.

This write-ahead logging system allows recovery/journal replay if a crash occurs during the commit process. Alternatively, if the system crashes before a commit, the log contents are discarded.

A side effect of this mechanism is that writes are not readable by the internal filesystem code until the corresponding transaction commits. This problem imposes limitations on internal filesystem code. For instance, any code that reads an on-disk state and then modifies it cannot be called more than once per transaction, or else it will read stale data.

Consistency Model

The journal logs all filesystem modifications from higher layers. As a result, our filesystem offers strong consistency along with atomicity guarantees. Much like in POSIX, after a file write goes through the commit process, any read of the previously written bytes will return the data specified by that previous write. Any new writes over that data will result in visible overwriting of that data from the perspective of other readers.

Note that this consistency model respects the limitation discussed previously. Highly concurrent operations across several clients may fail if users don't synchronize them to commits. For the predominantly single-or-few user edit-heavy workloads we expect, these issues do not come up frequently, if at all.

Inodes

We utilize a standard inode structure consisting of an inode number, reference count, filesize, list of addresses, and mode bit (file/folder). We deviate from a standard implementation that uses a direct/indirect addressing scheme by dropping indirect addressing entirely and using 511 direct blocks. We are aware that this limits the number of inodes (to only 16384) and the filesize (to ~1.99 MiB); our block-sized inodes are large enough that this is likely not an issue. Additionally, this gives us a minor performance boost and simplifies implementation as we do not need to navigate indirect block layers. These inodes give us the foundation required to construct our filesystem.

API

We provide the following POSIX-like API for user convenience:

```
type PP2 interface {
	Open(filename string) fd int
	Close(fd int)
	Read(fd int, count uint) string
	Write(fd int, data string) count uint
}
```

Our file system consists of a single root directory containing all files a user would like to store. We do not currently provide a way for the user to list all files in the filesystem.

Evaluation and Testing

We prioritize functional correctness, availability, and consistency over performance in our system. Therefore we do not evaluate performance extensively. However, we stress-tested our filesystem to the limits of each layer's capacity / concurrent transactions and found that the system offers regular performance up to these limits.

Our testing methodology to ensure correctness is multi-pronged. We first mock a non-distributed block layer and unit-test all filesystem layers built atop this mock disk. The unit tests cover each function in the absence of a network and other clients. Next, we integration test the POSIX interface in the presence of multiple clients, ensuring consistency guarantees hold while maintaining the reliability of the underlying network. Finally, to capture the possibility of network instability, we rerun these integration tests repeatedly in a stress test while randomly disconnecting servers and clients.*

Future Work

While our system is fault-tolerant and distributed, we did not optimize at all for performance, so an extension of this project would investigate ways to improve the speed of the underlying Raft implementation or seek viable alternatives. Additionally, our top-layer file system layout is extremely rudimentary, with only one directory. Another extension to this project would be implementing an intuitive way for users to visualize and navigate multiple directories for ease of use without violating our consistency guarantees. Lastly, an unmerged branch allows blocks to be stored on disk rather than entirely in memory, significantly reducing our RAM usage and realizing the filesystem's total 32 GB capacity.

**more details about integration tests can be found in our GitHub repository, in integration_test_procedure.md.*