David R. Ricardo
Electrical Engineering and Computer Science
Product Implementation Lead

Taylor H. Andrews
System Design and Management
Testing Verification & Performance Lead

MemoryFS: An In-Memory Replicated Fault Tolerant POSIX-Like File System

Motivation

In-memory filesystems have become widely implemented due to their performance gains for disk intensive workloads by offering an abstraction for a low latency, moderately sized filesystem that fits in RAM [1]. Relying on in-memory filesystems to store data for longer amounts of time could be considered risky for applications, as they normally believe the data they are persisting to "disk" is written to non-volatile storage. In the event of a power loss, in-memory file systems can be disastrous for losing system state, especially if the application is not aware it is using an in-memory filesystem to persist data as many common in-memory filesystems aim to appear as a generic mount [1]. More commonly, applications are using containers to scale, and an in-memory filesystem was considered as a possibility to replicate data to them efficiently, without having to make any disruptive host or container system changes. Our journey from this initial exploration took us through investigating, designing, building, and validating a system that replicates a simple POSIX-like in-memory filesystem across multiple machines using Raft to handle lossy networks, network partitions, and machine failure to ensure the in-memory filesystem stays available to clients.

Early Planning Stages

We started by exploring the fundamental interface POSIX offered to aim to support basic filesystem features such as creating files, deleting files, seeking, writing, reading files, and manipulating directories [2]. We then designed a spec for a network TCP/IP socket to allow the examination of what incoming communication would look like (see Appendix B); if a linux FUSE driver was selected to offer a generic mount point for applications to interact with, our system could then support a basic synchronized filesystem across generic Linux containers with the added benefit of being contained within the containers themselves, rather than depending on the host. We believe the design of the network interface first allowed us to see the problem from the actual context of how it would interact with the larger system around it, and it gave us a framework to start architecting the system with the right requirements in mind. We had large amounts of productive brainstorming, discussion, and whiteboarding to identify the counterpart components we would need to build and validate from Lab 3, and we both learned from one another's past experiences.

Challenges

Initial challenges revolved around a testing plan, and order of components to build. We knew the singlethreaded MemoryFS data structure would be much more complicated than the key/value store, and we recognized the need for it to have separate validation, before setting it afloat on Raft. David recognized the opportunity to establish a common interface to allow the reuse of what would become the core MemoryFS tests combined with different server numbers, lossy networks, and snapshots by generating very elegant wrappers. Taylor then automated the generation of bash scripts to run them all, creating simple pre-checkin testing, as well as workloads for a newly functioning Jenkins server Docker container, with an NGINX container serving the failing tests logs from his apartment to keep the project stable and to prevent regressions. We put the majority of validation focus on testing a combination of all the core MemoryFS tests to initially feel confident all filesystem state replication was happening reliably across Raft with and without lossy networks, different numbers of Raft replicas, and snapshotting, and then wrote some single specific tests analogous to the Lab 3 tests that simulate a network partition scenario.

More challenges revolved around scoping and finding reasonable goals we were likely to complete together with the time we had. After we had drawn out the whole system required to replicate data across Docker containers, we realized we could not start from opposite ends alone, and both had to focus initially on the design, implementation, and validation of the core filesystem data structure replicated across Raft. We still learned how Jenkins and Docker are deployed for real world software projects, and how Docker's volumes, image compilation, networking, and repositories work. We also located C source code for a simple FUSE driver tutorial project that would likely be used next to implement the thin layer that offered a generic linux mount point appearance, which would use a socket to drive our Raft Clerk, if a simple TCP/IP interface was added to send and receive the structs that currently are provided by the core MemoryFS test code [3].

Another set of challenges was establishing what the identity of an operation looked like in our system. Difficulties arose from using the operation itself as our operations byte slices were not immutable, which prevented it from being used as a key without further alteration. After some brainstorming, we settled on adding a timestamp field to the operation struct, in high hopes that then a quick SHA-1 hash of each operation that included those variable bytes would reliably produce a unique, immutable key to use for an in-progress map, when then allowed the reuse of patterns from Lab 3 code. We foresaw a hash collision in this area of the code would be disastrous, and most likely cause irreparable destruction to the replicated filesystem integrity.

Filesystem interface and testing

We spent considerable time deciding how many and which operations to support: too many and the project would be overscoped, and too few and it would be inefficient. We settled on seven: `open`, `close`, `seek`, `read`, `write`, `mkdir`, and `delete`. The first six correspond directly to standard C library functions, and `delete` acts as `unlink` on files and `rmdir` on directories. The specifications of these methods are exactly the same as the standard C functions, though in some cases we have reduced the set of optional arguments or errors. We chose these seven functions for usability: most of the experience of navigating a filesystem can be composed from these seven functions. We chose to omit `chmod` and related functions because security is not a priority and those functions would require great effort for little usability gain. In keeping with our priorities of simplicity and usability, we added the invariant that each file can be opened with only one file descriptor at a time. To enable atomic operations while also preventing unintentional livelock, we added a flag to `open` controlling the behavior when open is called on an already open file: it either blocks until the file is available or returns an error.

A final set of challenges revolved around utilizing 6.824 staff lab code to build tests for network partitions, and leader changes. We recognized the Put and Get functions from Lab 3 corresponded to a sequence of Open/Write/Close and Open/Read/Close functions on a file, and mirrored the common partition situation and leadership change that is simulated in TestOnePartition3A. In fact, we made tests based off every test from lab 3A, and they all pass reliably.

Performance Benchmarks

Performance challenges were encountered to determine what sized reads and writes offered the best Raft operation replication performance. The following table shows our performance numbers for replicating 10MB of data to the MemoryFS filesystem using various buffer sizes and numbers of writes for two different Raft configuration scenarios.

Buffer Size	# Writes For 10MB	Debug mode, 5 Host Replication, Lossy Network	Debug mode, 3 Host Replication, Reliable Network	Release mode, 5 Host Replication, Lossy Network	Release mode, 3 Host Replication, Reliable Network
10MB	1	22.55s	11.834s	**6.613s**	**5.845s**
1MB	10	21.079s	11.016s	7.670s	6.350s
512KB	20	24s	**10.627s**	9.389s	5.891s
256KB	40	22.16s	10.767s	11.844s	6.423s
128K	80	**19.5s**	13.830s	12.950s	6.600s
64K	160	29.711s	12.049s	23.623s	8.326s

We recognize this interesting best-performance behavior could further be automated, investigated, plotted, and extrapolated to potentially establish an optimized variable rate write buffer size protocol to ensure that client writes completed with the best performance depending on the host configuration, as well as if the network was known (or perhaps even detected) to be lossy or not.

After investigating the performance of how various sized buffers interacted with our debug code, we disabled all debug code and ran a 100MB write test using 10 writes with a 10MB buffer size. The data replicated across 3 hosts in 48.173s, indicating replication of data at 2.07MB/s. This is not out of line with the performance of our other tests, such as the 3 host replication on a reliable network with a 10MB buffer size, which transferred in 10MB in 5.845s for a transfer rate of 1.71MB/s.

Contributions

We present MemoryFS, an in-memory filesystem implemented in Go. We built off lab 3, a replicated key-value store in Raft, to replicate an in-memory filesystem. Along the way, we overcame many challenges because a filesystem is far more complex than a key-value store. We demonstrated that our filesystem can handle files up to 100MB efficiently and ran performance tuning to optimize write buffer sizes.

Citations

[1] Various. "Tmpfs." *Wikipedia*, Wikimedia Foundation, 10 Apr. 2018, en.wikipedia.org/wiki/Tmpfs.

[2] [Molay] Bruce Molay Understanding Unix/Linux Programming, Chapter 2, and [Stevens] W. Richard Stevens Advanced Programming in the UNIX Environment, Chapter 3. "FILE I/O." University of Chicago Computer Science, www.classes.cs.uchicago.edu/archive/2017/winter/51081-1/LabFAQ/lab2/fileio.html.

[3] Pfeiffer, Joseph J. "Writing a FUSE Filesystem: a Tutorial." New Mexico State University Computer Science, 4 Feb. 2018, www.cs.nmsu.edu/~pfeiffer/fuse-tutorial/.

Appendix A: How to run our code

GOPATH points to root of repository; export GOPATH=~git/6824-final-proj
Debug mode on or off: DFS_DEFAULT_DEBUG_LEVEL=0, 1, 2, or 3. 0 is no debugging and is the default.
src/test/run_precheckin_tests.sh will then execute all non-benchmark tests.
src/test/run_perf_tests.sh will then produce 10MB performance numbers.

APPENDIX B: An Initial POSIX-like interface spec for MemoryFS

We will have POSIX-like interface messages passed through a TCP socket to support a generic client. A generic message looks like the following:

```
Abstract PosixSockMsg (8 bytes + size of message payload struct below)
{
        uint8 posix_cmd_id = (1 byte - 8 bits - 255 vals for the Posix operation)
        uint8 _pad0[7]; //to start payload on a 64-bit aligned address
        ...
        <msg payload struct>
}
```

Replies will be have a return value (which will hold success data like file descriptors) and an error code for errors, which we will begin to enumerate and prioritize lower.

```
struct PosixMsgReplyInt
{
        int return_val; //file_descr, -1 on error
        uint8 error_code; //operation specific but 255 vals is enough
}
```

We can also have a reply with just a byte worth of return values.

```
struct PosixMsgReplyByte
{
        uint8 return_val; //0 success, -1 on error
        uint8 error_code; //operation specific but 255 vals is enough
}
```

Below we enumerate structures for the common Posix file operations we aim to support:

```
struct OpenMsg (open cmd specific struct) (4 bytes)
{
        uint8 mode = {O_APPEND,O_CREAT,O_TRUNC}
        uint8 flags = {O_RDONLY,O_WRONLY,O_RDWR}
        uint8 path_len; //length of path in chars / bytes
        uint8 rel_path[512]; // 512 char file path limit selected
}
struct OpenMsgReply  (abstracts to PosixMsgReplyInt) (4 bytes + 1 byte = 5 bytes)
{
        int return_val = {file_descr,-1 on error} //FD must be lowest available
        uint8 error_code = {ENOTDIR,EACCES,EISDIR,EMFILE};
}
```

Notes:
ERROR CODE - 1 byte - 8 bits - 255 vals

The error code will be used to communicate to our software about any issues that we ran into. 255 different errors should be enough, man 2 open page lists 30-something errors. Top 4 errors prioritized. Error numbers TBD as we figure out the errors we actually need to support.

```
struct CloseMsg (close cmd specific struct) (4 bytes)
{
        int file_descr;
}

struct CloseMsgReply (abstracts to MsgReplyByte) (2 bytes)
{
        uint8 return_val = {-1, 0} //{err, success}
        uint8 err_code = {EBADF, EINTR}
}
```

Time to seek to a place in a file to read/write!

```
struct SeekMsg (4 bytes + 4 bytes + 1 byte = 9 bytes)
{
        int file_descr;
        int offset;
        uint8 base = {SEEK_SET, SEEK_CUR, SEEK_END}
}

struct SeekMsgReply (abstracts to MsgReplyInt) (4 bytes + 1 byte = 5 bytes)
{
        int return_val = {-1, offset from beginning of file} //{err, success}
        uint8 err_code = {EBADF, EINVAL, ESPIPE}
}
```

An interesting semantic is that an offset outside of the current size of the file results not in an error, but in a “hole” being created with \0s on the next write. Cannot offset before beginning of file. lseek(file_descr, 0, SEEK_CUR) returns the current offset. We probably/hopefully do not need to support the check for pipes, FIFO, or socket semantics.

Read is where we get into returning real data.

```
struct ReadMsg (4 bytes + 4 bytes = 8 bytes)
```

```
{
        int file_descriptor;
        int nbytes;  // will govern message size
}

struct ReadMsgReply (4 bytes + 2 bytes = 6 bytes)
{
        int return_val = {-1, nbytes read} //{err, success}
        uint8 err_code = {EBADF,EFAULT,EAGAIN,EINVAL,EIO}
        uint8 buf[1GB]; // up to 1GB file read support… ?
}
```

On to writing data!

```
struct WriteMsg (4 bytes + 4 bytes + up to 1GB = up to 1GB)
{
        int file_descriptor;
        int nbytes;
        uint8 buf[1GB]; // up to 1GB file write support…?
}

struct WriteMsgReply (abstracts to MsgReplyByte) (2 bytes)
{
        uint8 return_val = {-1, nbytes written} //{err, success}
        uint8 err_code = {EBADF, EFAULT, EINVAL, EFBIG, ENOSPC, EAGAIN, EINTR, EIO}
}
```

On to creating new files! Some funny business with it requiring an int creat(..) prototype but it is implemented with an open(..) call.

```
struct OpenMsg (creat cmd specific struct) (3 bytes + <=512 bytes = <=515 bytes)
{
        uint8 flags = {O_RDONLY,O_WRONLY,O_RDWR}
        uint8 mode = {O_APPEND,O_CREAT,O_TRUNC}
        uint8 path_len; //length of path in chars / bytes
        uint8 rel_path[512]; // 512 char file path limit selected
}
```

Finally duplicating files:

```
struct DupMsg (dup cmd specific struct) (4 bytes + 4 bytes = 8 bytes)
{
        int old_file_descr;
        int new_file_descr; //unused if we don’t implement dup2
}
struct DupReplyMsg (abstracts to MsgReplyInt) ( 4 bytes + 1 byte = 5 bytes)
{
        int return_val = {-1, value of newd} // {err, success}
        uint8 err_code = {EBADF, EMFILE}
}
```

And deleting:

```
Struct UnlinkMsg
{
        string path;
}
```

```
Struct UnlinkReplyMsg (abstracts to the standard reply)
{
        int return_val = {-1, 0} //{err, success}
        uint8 err_code = {EBADF, EINVAL, ESPIPE}
}
```

Citations