Peter Zinkovsy (peterz3), Michael Kokkines (mgk3), Martynas Juskelis (mpj4)
Code: https://gitlab-beta.engr.illinois.edu/cs425-peter-michael-marty/mp2/

MP2 Report: Simple Distributed File System

System Design

Overview

We created a distributed file system. We use gossip for failure detection, and gossip pings occur over UDP. We use TCP for file system communication. At each node we maintain a map of filenames to file metadata objects. Each file metadata object contains the size of the file, a list of the servers that store the file, and various other flags used by the locking system (see below).

1. When **PUT** is called, we find the four servers with the most available memory. We then send a fixed size header (which includes file size, file name, etc), and the file bytes over TCP to each of these servers. Finally we notify all the nodes that a file has been added.
2. When a **GET** is called, we find a server that has the file, and request for it to send it to the local server. We then copy it to a local directory.
3. **DELETE** operates similarly to PUT, but it only sends a header -- not the file data.

Locking

Each file metadata object contains a boolean flag indicating whether the file is being written and an integer flag storing the number of machines currently reading the file.

1. The write function blocks until no other machine is reading or writing the file. The machine will then notify all the other machines that it is attempting to write, which will then lock the file and send a response. Once all the responses are received, the original machine executes the write, unlocks the file, and pings the other machines to unlock it.
2. The read function blocks until the file is not being written. It then pings the other nodes, which increment numReaders and send a response. After receiving all responses, the machine executes the read and pings the other machines to decrement numReaders.

Failures

When a node fails or leaves, the active node with the smallest index handles replication. For each file on the inactive node, it finds two servers: one that has the file, and the server with the lowest used memory without the file. It then sends a request to the first server to duplicate the file onto the second server. If the handler crashes, the node with the next lowest index takes over.

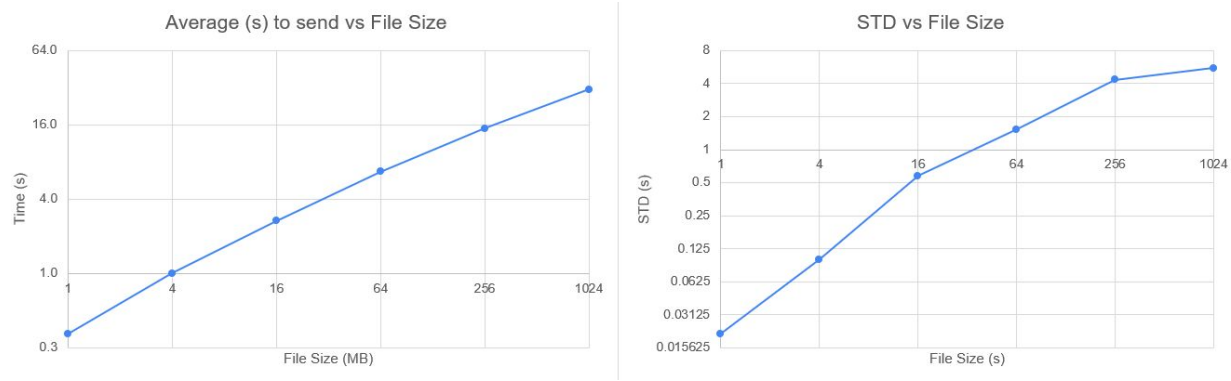
Graph 1 Explanation

Note that the axes on this graph are logarithmic. In general, sending 4x the file size takes 2x as long. We are using Go's IO library to write and read file data to the TCP connection. Our hypothesis is that this library contains optimizations that improve the runtime as the file grows

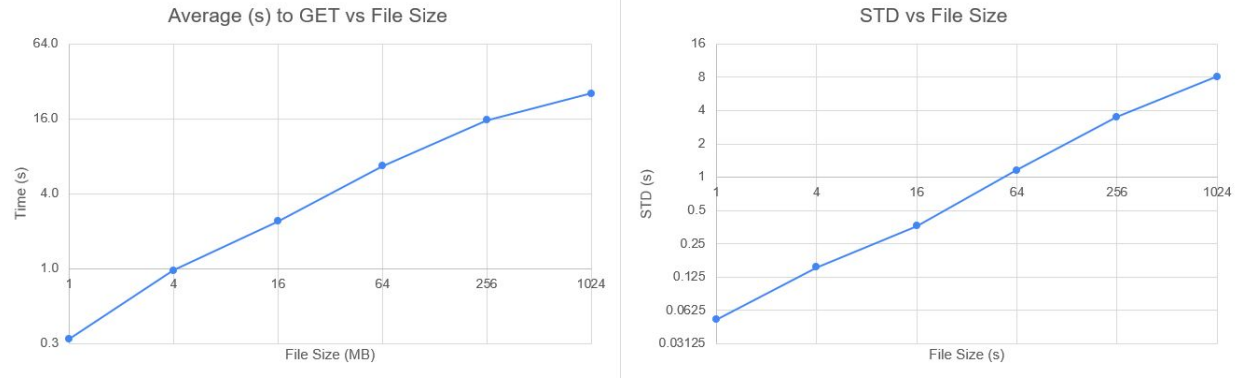
Graph 2 Explanation

The get graph is similar to the send graph, which is expected as the bottleneck is sending the data over TCP. In general it was faster, which makes sense as it does not have the overhead from blocking.

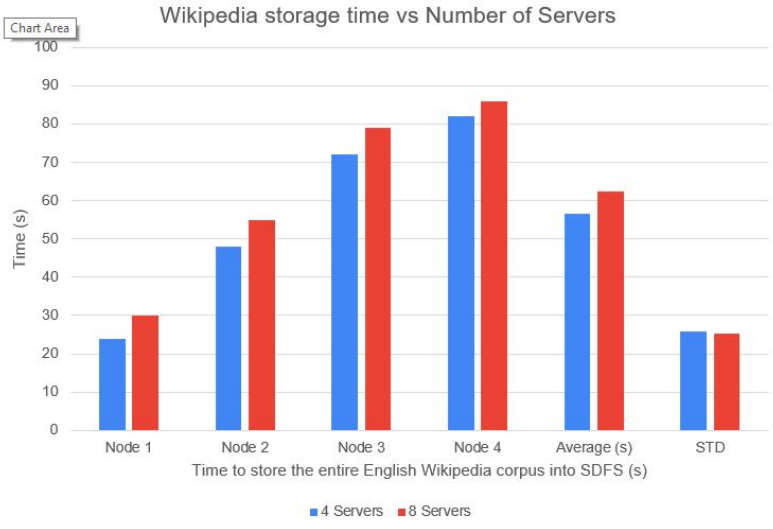
Graph 1



Graph 2



Graph 3



Graph 3 Explanation

This graph depicts the total time elapsed from the PUT command to a specific node receiving the full file. Since we block on writes, sequential timings are expected. To be fault tolerant through 3 failures means we only need 4 copies, which is why there is little difference between the times on 4 servers vs 8 servers - exactly 4 files are copied in both cases. 8 Servers are expected to take slightly more time as it consumes more resources to maintain a larger list of nodes.