

FA20 CS425 - Distributed System Lab Report 2

Yichi Zhang, Ali Zaidi, and Bo-Rong Chen

System Design

Overview. We implemented a distributed file system in this MP. In order to maintain a consistent file list across the system, we use leader election and a consistent hashing ring to elect a leader and make decisions regarding file operations, including **GET**, **PUT**, and **DELETE**. The election algorithm we used here is ring election; we implemented a consistent hash ring to map files onto their respective nodes. Once a node is elected as a leader, it will be the only node in the system to modify the global file list (*i.e.*, a map of `sdfsfilename` to all nodes that hold its replicas). Since we are using the all-to-all heart-beating protocol, we will know that once the leader has failed, we can initiate a new round of election. All file operations are ignored until the next leader is elected.

For a **PUT** operation, an initiating node A sends a query to the leader, who will reply with a replica destination B for A to transfer the file to based on the hash ring. After the first transfer is completed, leader will update the global file list to include A and B and ask both A and B to transfer the file to other two replica destinations C and D *in parallel*, in order to speed up the **PUT** operation. Once these file transfers are completed, the leader will update the global file list to include nodes A, B, C and D.

For a **GET** operation, a request will be sent from initiating node A to the leader, which follows the similar design as **PUT** operation but with a *reversed* sending path. TCP is used for leader election and file operations, and our UDP implementation from MP1 is used for heart-beating and failure detection.

Failure Detection and Re-replication. In case of node failure we first check if it is the leader. If so, we will initiate another round of election and proceed with the following steps: the leader should check which files are stored on that machine and re-replicate them using the same method as **GET** operation. Furthermore, if the new leader is elected after the previous leader fails, it will check if the number of replicas is greater than 4, if not, it will re-replicate the file up to 4 replicas. Moreover, the leader also checks if the failed node is the sender of a **GET** or **PUT** operation. If so, the leader will execute the requests again.

Results

The curves of both **GET** and **PUT** are linear in terms of the file sizes. **PUT** will take two passes: the first pass is for replicating one file; and the second pass is for these two nodes to replicate the file into other two nodes, whereas **GET** will only send the file from one node containing the requested file to the target node. This takes only one pass. Therefore, the **PUT** latency is nearly double compared to the **GET** latency, as shown in Figure 1. In addition, the design of **GET** is the same as what the leader does for file re-replications after a failure, and it takes half of the latency to do **PUT** of the same file in our system.

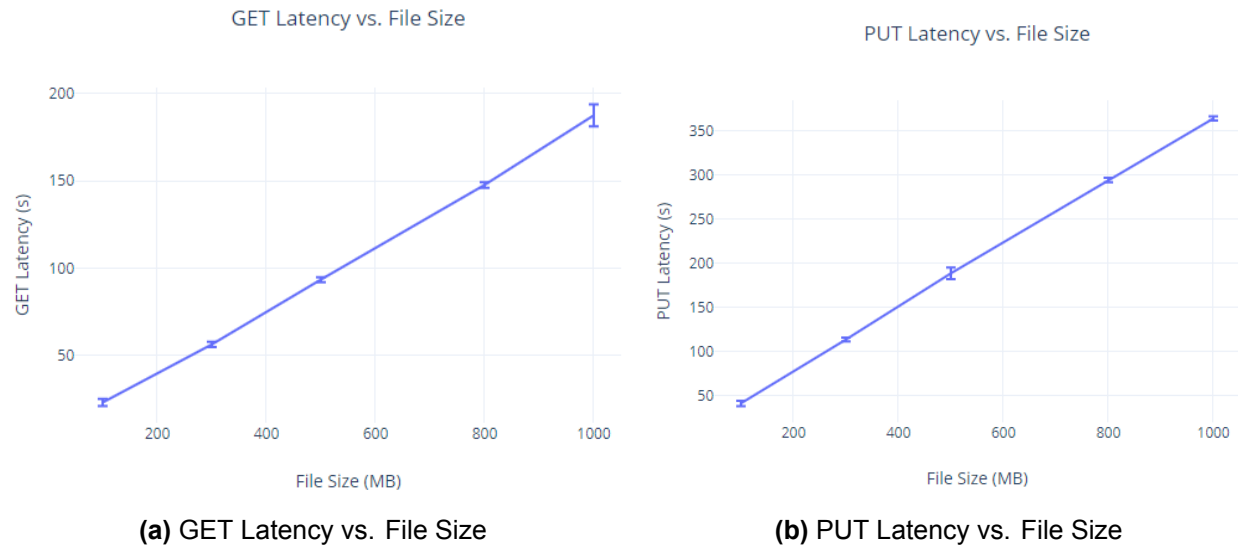


Figure 1: Latency in terms of different File Sizes

Wikicorpus PUT Time vs. # of Machines

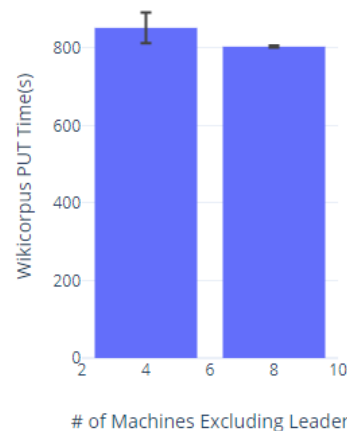


Figure 2: Store English Wikipedia corpus

Moreover, we have tested our system under transferring large files, as shown in Figure 2. The latency of putting the entire English Wikipedia corpus (*i.e.*, 1.3 GBytes) to our system containing 4 and 8 machines are about 852.14 seconds and 803.64 seconds, respectively. This shows that our **PUT** operation is independent of how many nodes are in our system.

Our results show that the throughput of **PUT** is roughly 12 Mbps.