MapReduce + Peer-To-Peer Data Replication System

Aayush Joshi, Gabriel Rodriguez, Tymon Vu

June 7, 2025

Abstract

For our final project in CS569, we implemented a simplified distributed peer-to-peer data replication system that was inspired by early file sharing platforms like Napster, more specifically replicating files using popular peer-to-peer file sharing systems [1], and the torrent protocol. Just like we have been doing with other projects in the class, each node will be represented by a Go process that acts as both a client and a server, enabling direct communication between peers to request and replicate files. The project originally builds off our original implementation for Lab 4 where we added in leader election and task assignment in order to complete MapReduce on some given text files. Our implementation has the system design stay functional if any member of the cluster leaves or joins at any time. The system demonstrates core distributed computing concepts we've been learning throughout the class, such as file chunking, fault tolerance, and decentralized coordination, resulting in a lightweight and scalable model for data sharing.

1 Introduction

MapReduce was a paradigm introduced back in the early days of Google's inception to deal with the parallel processing of its different products. It was greatly utilized for PageRank, Google Maps, and clustering articles. It has two distinct stages based on its name, the map stage and reduce stage. The mapping stage sees the big task at hand divided into multiple parts for simpler processing across the nodes. This is all handled with an elected leader through Paxos consensus protocol. In the mapping stage, nodes would map the data into key-value pairs which then gets passed off to the reducing stage once all the data is mapped. The reducing stage tasks the different key-value pairs and "reduces" them into a singular instance of all the keys and have the value as the number of times the key appear. This system is very useful for getting the word count of large text files or chunks of data that is difficult for one machine or processs to parse through individually.

Our implementation of MapReduce follows this system by basing it on our previous labs. Gossip and Raft have enabled us to keep track of our nodes, their liveliness, and passing information between them. Our previous lab had us implement leaders through elections and now MapReduce gives the leader the task of assigning tasks to other nodes and being the source of truth for tasks. With our implementation of MapReduce, we keep track of the state of the different tasks with "task-log.jsonl" (Pantoja approved this). While also writing down the results of the key-value mapping to files mark "mr-tasknum-seqnum" to be later used to reduce down further. If any workers die while on the job, we can sweep them under the rug and have another take their place and finish the job. In the case of a leader dying, there is a break in MapReduce to elect another and proceed to finish the tasks, if all of them are not finished at this state. Due to the quick nature of the jobs and the relatively small amount of data initially given, we decided to add a few more free books from the Project Gutenberg library to the ./data folder to give us a bit more insight on how scalable our results seem and if there is any discrepancies from the original intended given MapReduce boilerplate code.

After MapReduce finishes all of its tasks, peer to peer data replication would happen. We tried to replicate "log replication" by copying the output of our combined MapReduce word count from all the given text files to each of our node's respective output files. Essentially, the leader node would broadcast each

key-value pair in its combined output to all the other nodes, and once these follower nodes pass a certain threshold of data entries, they would eventually write that information down in their respective log. If at any point a failure occurs where the follower nodes missed some information from the leader or logs get out of sync, our peer-to-peer implementation would take over to take the load off the leader to get the node back up to speed. Overall, our implementation prioritizes keeping up follower node log replication to match the broadcasts of the leader, checking and accounting for gaps in entries if any missing information is present.

2 Design Description + Implementation

Building on the foundation from previous labs, we implemented additional logic to enable peer-to-peer data replication across all nodes. This enhancement introduced a set of new functions that handle communication between nodes, manage data consistency, and ensure fault tolerance during replication. The following sections describe the key components and mechanisms we developed.

2.1 Function Breakdown

- **AssignData**: Called by the leader to split data distribution among the other nodes in the cluster. Splitting is done by line number and not by size. Error handling includes setting negative values if a node is unsure about line number.
- **BroadcastData**: A leader function that reads the next DataID line from the map-reduce final output (`mr-out-final`) and broadcasts it to all nodes. This is done iteratively every two seconds.
- **SendData**: Called by a worker node to send a specified range of data entries to another node. Data is retrieved from memory if available, otherwise read from disk. It ensures the response includes the full range requested.
- **ReceiveData**: Called by any node to accept and insert received data into its local state. It supports both broadcasted and targeted data, updating local memory and triggering the dump process when necessary.
- **CheckAndDump**: Periodic self-check function for a node. It identifies missing data ranges in memory and disk, requests re-replication from the leader, and dumps data to local disk when a threshold is met (e.g., 5+ in-memory entries).
- **dumpDataToFile**: Serializes the in-memory `DataItems` into a JSON file named after the node. It merges with any existing file contents and overwrites them atomically.
- **ReadDataById**: Reads and returns a range of `DataItem`s from the local disk file using the data map stored in the node's replication file.
- **ReadDataFile**: Static function used by the leader to read a specific range of lines from `mr-out-final`, the final result file of map-reduce. Returns raw string lines.
- **insertDataItemSorted**: Inserts a `DataItem` into the local slice in sorted order by ID. It replaces existing entries with the same ID.
- **readDataMapFromFile**: Reads and decodes the node's replication JSON file into a map of ID to string values. Used for checking existing data before reading or writing.
- **PrintDataItems**: Utility function for debugging. Prints all current in-memory `DataItems` in sorted order to stdout.

2.2 Pipeline

The system begins by executing a MapReduce job (getting the word count of given text files) that will produce an important mr-out-final file, which combines all the MapReduce output data from all the other worker nodes. This file serves as the initial source of data entries for replication, where each line corresponds to a key-value pair, where the key is the word that is being counted and the value is the frequency of the word in the given data set.

Afterwards, the leader node initiates replication by broadcasting new data entries to all follower nodes. This broadcast is scheduled to run at fixed 2-second intervals, with each transmission containing a single new (DataID, Value) pair. The DataID is the respective line number in the output file we are, which will be used to check whether a node is behind in replication or not, and the value is the actual content of the line (the key-value pair from MapReduce). Upon receipt, each follower updates its in-memory data structure to include the new entry Once a node reaches subset of 10 buffered entries, the node will write to their own file named ¡nodeid¿-replication.json. This checkpointing mechanism allows a node to recover its state on restart without requiring a full replay of the entire data history from the leader.

2.2.1 Follower Death + Restart

If a node were to join the cluster late or on restart, it requests the 1st item and all other in-memory items to begin with. This is done because every 10 seconds, the CheckandDump function is called that checks both in-memory and the local json to see if there's any gaps of data missing from the leader. If it recognizes that any gaps are missing, it will request these entries, but it will not dump the data if it finds these missing gaps. Instead, once it receives all pending data, it will dump it on the next scheduled call to CheckandDump.

2.2.2 Leader Death + Restart

In the case of leader failure, the system falls back to a re-election protocol that selects a new leader, and replication would continue off the last known DataID. The correctness of resuming replication from the new leader depends on the assumption that the elected node holds the most up-to-date DataID and term, which is accurate because the new elected leader will have an updated term number and therefore an updated DataID.

3 Results

Overall, our peer-to-peer log replication system performed reliably, successfully replicating data across nodes even in cases where the leader was unavailable. When a node failed to replicate directly from the leader, it was able to communicate with peer nodes to recover the missing data and correctly dump its in-memory state into its own local file. However, even better was that it was able to successfully be implemented on top of our MapReduce logic without issue, as explained in the previous section to create a seamless pipeline from working on MapReduce to replicating the output log from the leader across to all the alive worker nodes.

4 Scalability

Considering scalability beyond the multiple nodes we've structured for our labs, we share a lot of similarities with other traditional distributed systems like Napster and Gnutella. Where Napster's centralized index model simplifies data discovery at the cost of bottlenecks and points of failure and Gnutella's decentralized approach scales well horizontally but adds a bit of overhead. We approached it by taking the middle ground based off our previous lab work. By having a central leader for task management and log replication we do introduce certain bottlenecks but offset that choice by leveraging our peer to peer connections for recovery for our follower nodes. Scalability in our system is achievable by adding more nodes and distributing the leader workload. However, we could face challenges with extreme scalability where network traffic

and coordination complexity comes to back to bite us. Future improvements focusing on load balancing, optimized broadcasting, or even role adjustments could sustain better efficiency at larger scales.

5 Individual Contributions

- **Aayush**: Worked on data broadcasting functionality carried out by the leader, peer-to-peer data replication assigning and distribution in case a node fails or joins the cluster late and node's in-memory and local data management
- **Gabriel**: Worked on initial MapReduce and lab's implementation that was used as a basis for the initial peer replication. General logic planning for the peer replication and node behavior. As well as writing and editing the final report.
- **Tymon**: Worked on initial MapReduce and Paxos implementation to to elect leader, which was basis for initial peer replication. Organized files in filesystem to the way they would flow and node behavior to make sense and final writing and editing in the final report.

6 Conclusion

Our implementation successfully demonstrates the synergy between MapReduce and peer-to-peer data replication. By leveraging foundational principles from established distributed file-sharing systems like Napster and Gnutella, we managed a robust solution capable of fault tolerance, efficient node recovery, and data consistency through decentralized coordination. Each node's ability to dynamically assume roles and seamlessly continue operations despite failures highlights the reliability and efficiency of our design.

References

[1] Saroiu, S., Gummadi, P. K., and Gribble, S. D. Measurement study of peer-to-peer file sharing systems. In *Multimedia computing and networking 2002* (2001), vol. 4673, SPIE, pp. 156–170.