The Hitchhiker's Guide to Building a Distributed Filesystem in Rust. The beginning...

It all started after I started to learn **Rust** and picked a learning project to keep me motivated, it was an **encrypted filesystem** `https://github.com/radumarias/rencfs`. There I got the basics on writing a filesystem with **FUSE**, **encryption**, **WAL** concept, **data integrity**, **parallel processing**, **filesystem internals**.

The next challenge I picked was building a **distributed filesystem**. Was always fascinated by distributed filesystems, using **Hadoop (HDFS)**, **Spark**, **Flink**, **Kafka**. I became familiar with the concept of **sharding** using **Elasticsearch**, with clusters leader and election process using **MongoDB**, with **WAL** from **PostgreSQL**. The next phase was collecting a lot of links to read about how to build it. After a period of research I ended up understanding the basic concepts and selecting some frameworks to use. I concluded to the following structure and frameworks for the system.

COORDINDATOR NODES. These will be the entry points to the system for the client apps. They will be responsible of creating the **structure**, saving the **metadata** and create **logical distribution** of the **shards** (chunks from files). They will be served with **gRPC** using **Apache Arrow Flight**. They will run in a distributed **Raft** cluster or if we don't want the penalty of a **single active master** at a time we can use smth like **CRDTs** (Conflict-free replicated data type) with **Redis Sets** ensuring the **contraints** like **uniqueness of file names** inside a folder. For actual **sharding and distribution** will split the file in shards of **64MB** and using **Consistent hashing** (used by **MongoDB** and **Cassandra** partially) or **Shard keys** (used by **tikv** and **Cassandra** partially) will distribute the shards along with their replicas on multiple data nodes.

Quick explanation on how Consistent hashing works. We **hash** all node **names** or **IPs** and we create a **ring** with points in interval $0 \dots 2^{64} - 1$ from the hashes values. We add **v-nodes** which are virtual nodes, built from nodes names with sequence suffix, so the distribution of hashes is more evenly distributed. We'll use **BLAKE3** for hashing. Then we hash the **file key**, like the absolute path or some unique identifier, we have a number on the ring corresponding to the hash and we search the closes node hash clockwise using binary search **O(logn)** or linear search **O(n)**. That will be the node where the shard will be placed. We do this for all replicas also, which are hashed from the key adding a sequence suffix, removing already assigned nodes from the ring as we distribute next replicas. Both sharding techniques works good when nodes are added (when we need more space) or nodes are removed (when they have failures or we want to reduce the costs). **Consistent hashing** redistributes the shards when nodes configuration changes, **Shard range** split existing ranges and assigns them to new nodes and merge ranges with existing ones when nodes are removed.

The metadata will be saved in **tikv** DB and communication with data nodes will be via **Kafka** (each node will have its **topic**) to avoid **congestion**, have **decoupling** and **retries**. Coordinator nodes will keep a list of **ongoing tasks** for the data nodes and in case a data node dies it will reallocate the operations, shards and tasks to another nodes.

DATA NODES. Once the structure and **logical distribution** finishes the client communicates directly with data nodes to **upload**/**download** the shards. This will be made via **HTTP** with **Content-Range** header at first and **BitTorrent**, **SFTP**, **FTPS**, **gRPC** with **Apache Arrow Flight** or via **QUIC** later on. After a shard was uploaded, we check the transferred data with **BLAKE3** and also we check it after we save it to disk, in order to ensure **data integrity**. First we write data to a **WAL**(Write-ahead logging) and then periodically or after all file has been uploaded we write the chunks to disk. This strategy is widely used by DBs to ensure integrity as if the **process dies** or we experience a **power-loss** while writing, next time we restart we continue the writing until all changes are applied.

FILE SYNC. We will implement **Mainline DHT** which will be an interface for **DHT query** and will read the data from **tikv**. This eliminates the need of a **tracker**, which is a **single point of failure**, the **DHT** is a **distributed Hastable**. Then we sync the file content via **BitTorrent**, this makes sense because we will have multiple **replicas** for each file so a node can read from multiple **peers**. Plan is to implement the transport layer with **QUIC** and take advantage of **zero-copy** with **sendfile()** which sends the data from disk directly to socket, without going through the **OS's buffer not using CPU**.

FILE CHANGES. After the file is synced we will create a **Merkle tree**, and when the file is changed we just compare the trees starting from the root, between the nodes to determine the exact part and chunks from the files which got changes so we sync only those.

CAP THEOREM: We target especially **Consistency** and **Availability** when possible.

NODES FAILURES. No distributed system **I have experiencd is** 100% **fault free**. We need to prepare the system for failure and adapt in such cases when nodes goes down. There are different types of failures and strategies what to do in such cases but we will be prepared for these kind of failures: **Node**, **Network**, **Software**, **Partition**, **Byzantine**, **Crash**, **Performance.** This is how we can make **failure tolerant** systems: **Redundancy**, **Replication**, **Graceful Degradation**, **Fault Isolation**, **Failure Detection**.