Introduction

In this article, we propose a blockchain network that acts as a centralized append-only

distributed file system (DFS) such as Hadoop Distributed File System (HDFS) or Google File System (GFS). The potential advantages of blockchain as a distributed file system (BaaDFS) include:

- High availability

: a user can tolerate failures of ⅓ full nodes and enjoy almost 100% high availability on read unless all nodes are down without worry about the single-point of failure of metadata servers (such as namenodes in HDFS) of traditional systems;

- High data integrity

: a node could fully validate the integrity of any piece of the data written to the DFS. The validation can be very efficient: given the position of the data (filename, offset, len), the cost of validating the data has the same order of the cost of client read operation.

- Highly trustworthy storage for clients

: a writer can check the integrity of the written data and ensure immunity, and any reader could verify such integrity and immunity.

Currently, the blockchain network is designed for private use (as the target DFS is centralized), but it should be able to extend to public/consortium use with some modifications.

DFS Semantics Supported

The BaaDFS supports single-writer multiple-reader append-only DFS semantics, which has been widely used in existing DFSs such as HDFS and GFS. To be more specific, for a given file, we only allow to write from a writer at any time, but multiple readers may open and read the same file simultaneously. The writer could only append data to a file. Such DFS semantics is highly suitable for high-performance batch processing.

Client Operation Semantics

The BaaDFS supports the following file operations from client perspective:

- create(filename): create a file for write operation, return a file object upon success or throw if filename exists or other IO exception.
- open(filename, isReadonly): open a file for read or read/write, return a file object upon success or throw if filename is not found or other IO exception.
- read(file_object, buffer, offset, len): read the data of the file starting from offset and up to len bytes to buffer. Return the number of bytes read or -1 if the EOF is reached.
- append(file_object, buffer): append the data from buffer to the end file. Throw if a concurrent append is detected.
- flush(file_object): flush all cached append data to the network, and wait until the data are visible to all readers. Throw if concurrent append/flush is detected.
- close(file_object): flush if necessary and close the file.

Representation of the File System as a Blockchain Ledger

Similar to HDFS/GFS, a file in BaaDFS is represented as a list of data chunks

where the file content is equally divided into chunks with the same size (chunk_size) except the last chunk, whose size, namely, last_chunk_size, is file_size % chunk_size.

In the proposed BaaDFS, a chunk of data is stored as part of a blockchain block, where the block consists of

- a header
- a list of write transactions, where each transaction is

tx := (filename, chunk_idx, chunk_num, chunk_data_0, chunk_data_1, chunk_data_${chunk_num - 1}),

which means that the data chunk_data_0, chunk_data_1, …, chunk_data_${chunk_num - 1} are written to the file with the offset starting from chunk_idx * chunk_size. The size of chunk_data's in a transaction must be chunk_size, except the last one, whose size, last_chunk_size <= chunk_size. The resulting new file_size after applying the write transaction becomes (chunk_idx + chunk_num - 1) * chunk_size + last_chunk_size. The hash of the list of transactions (likely in a Merkle tree way) will be stored in a field of the block header as conventional blockchain does.

Note that since we implement an append-only file system, the write transactions must satisfy the following constraints (assuming file_size' is the pre-write file size, and file_size is the post-write file size)

- (Chunks unchanged except the last one) chunk_idx > file_size' // chunk_size (// is the integer division operator)
- (Last chunk write must be append) if chunk_idx == file_size' // chunk_size + 1, then chunk_data_0 must contain last_chunk_data', where last_chunk_data' is the last chunk of the file before the write transaction.

To lookup the chunk, we define a chunk_info, which tells where the chunk data can be read as

chunk_info := (block_index, tx_index, tx_chunk_index)

where block_index is the height of the block that contains the corresponding write operation/transaction, tx_index is the position of the write transaction in the block, and tx_chunk_index is the position of the chunk_data in the transaction.

As a result, given the history of the ledger, i.e., blocks, a reader could fully read any part of the file by a list of chunk_info's together with file_size and chunk_size, where the list of chunk_info can be efficiently implemented as a Merkle tree (likely an accumulator) for fast update (only the last item), append, and read.

The tuple of (file_size, chunk_size, and chunk_info_trie_hash) is defined as the metadata of the file as:

metadata := (file_size, chunk_size, chunk_info_trie_hash),

and the state of the ledger given a block is basically a mapping as:

state := filename -> metadata

which could be implemented as another Merkle tree (e.g., Patricia Merkle Tree or Sparse Merkle Tree), whose hash value will be stored in the header of the block.

Summarizing the aforementioned details, the diagram of the ledger of a BaaDFS looks like

[

1600×954 87.4 KB

](https://ethresear.ch/uploads/default/original/2X/c/c551727192e148ba7511f4b971e945cdd6606a45.png)

A good property of such a file system representation is that given the hash of the state trie or a block hash, a reader or writer could uniquely determine a snapshot of the filesystem and check the integrity or immunity of the filesystem.

Components of the BaaDFS Network

In this subsection, we illustrate a blockchain network and its components for the BaaDFS

- Full node

: A node that maintains a replica of the blockchain ledger. For better performance, the node may be implemented as a cluster - a group of servers that act as a single node in the network. The full nodes are connected via a network protocol (P2P or membership managed by a configuration service such as ZooKeeper). Upon observing a new valid block is produced, it will synchronize the block from its peers and append it to the ledger. The nodes may not necessarily trust each other in the network.

- Consensus

: We will employ a fast-finality consensus, which may be Paxos/Raft for private/consortium usage or Tendermint for consortium/public usage.

- Finalized block:

A finalized block is that a block that reaches finality and will be irreversible by the consensus. This means that if a block is finalized, all the write transactions of the block (and previous blocks) and the resulting file content, i.e., bytes in offset [0, filesize], will be immutable and be consistent among new readers. Furthermore, we denote the finalized block with the highest index and its index as LAST_FINALIZED_BLOCK and LAST_FINALIZED_BLOCK_INDEX, respectively.

- DFSClient

: A DFSClient is a client run by a user that performs the interface of aforementioned file operations and translates the operations to blockchain operations. It will connect to one or multiple full nodes in the network.

Example Implementations of Supported Operations

We assume a full node has the following RPC service to a DFSClient:

- QUERY_FINALIZED_INFO: Query and return LAST_FINALIZED_BLOCK or LAST_FINALIZED_BLOCK_INDEX;
- QUERY_METADATA: Given a filename and a block index/hash, return the metadata of the filename of the block;
- QUERY_CHUNK_INFO: Given a chunk_info_trie_hash and chunk_index, return a chunk_info;
- QUERY_CHUNK_DATA: Given a chunk_info, return a chunk data;
- SUBMIT_TX_AND_WAIT: Given a transaction (tx), return success if the tx is included by a block before or equal to FINALZED_BLOCK, otherwise error.

Given above RPCs, the file operations can be supported by a DFSClient in BaaDFS as follows:

- open(filename, isReadonly): The DFSClient will look up the metadata of file in the LAST_FINALIZED_BLOCK from the network, store the metadata in a file_object, and return the file_object.
- read(file_object, buf, offset, len): The DFSClient will look up the chunk_info's associated with the chunk_info_trie_hash from the cached metadata and the read range of the file. For each chunk_info, DFSClient will read the corresponding chunk data, write the data to the user-provided buffer, and return to the user.
- write(file_object, buf): An optimized DFSClient will likely cache all contents from a write operation in the internal buffer and return immediately. After collecting several full chunks, the DFSClient will issue an actual write tx. This will reduce the number of writing tx on the last chunk with size < chunk_size, which requires read-modify-write and could be expensive.
- flush(file_object): This will forcibly issue write transactions and synchronously wait until the transaction is finalized (i.e., included by a finalized block).
- close(file_object): The operation will call flush(file_object) and destroy the file_object.

Advantages over HDFS/GFS

- High availability

: Consider the network adopts a BFT consensus that tolerates up to f byzantine failures with 3f + 1 full nodes

- The network can continuously perform writing even f full nodes have byzantine failures;
- For consistent read, as long as the DFSClient reads the metadata of the file in the LAST_FINALIZED_BLOCK from the network and read the content of the file from a full node with LAST_FINALIZED_BLOCK, the network will continue serving reading. Suppose a reader needs to read at least f + 1 full nodes to determine LAST_FINALIZED_BLOCK, read operation can tolerate 2f byzantine failures;
- If the consistency of read can be relaxed, the DFSClient may just read the metadata of the file in the last FINALIZED_BLOCK from any node and read the content of the file from the same full node, the network will continue serving reading until all full nodes are down.

- The network can continuously perform writing even f full nodes have byzantine failures;
- For consistent read, as long as the DFSClient reads the metadata of the file in the LAST_FINALIZED_BLOCK from the network and read the content of the file from a full node with LAST_FINALIZED_BLOCK, the network will continue serving reading. Suppose a reader needs to read at least f + 1 full nodes to determine LAST_FINALIZED_BLOCK, read operation can tolerate 2f byzantine failures;
- If the consistency of read can be relaxed, the DFSClient may just read the metadata of the file in the last FINALIZED_BLOCK from any node and read the content of the file from the same full node, the network will continue serving reading until all full nodes are down.

- High data integrity

: Given the position of the data to be validated (filename, offset, len), the node could validate the integrity of the data as follows (assuming the LAST_FINALIZED_BLOCK and its hash is valid (agreed by consensus)):

1. Read the metadata of the file from the state_trie_hash of LAST_FINALIZED_BLOCK, and validate all cryptographic proofs that the metadata is indeed in the state_trie.
2. Read the chunk_info's from the chunk_info_trie_hash in the metadata, and validate all cryptographic proofs that the chunk_info's are included in the chunk_info_trie.
3. Read the data chunk for each chunk_info, and verify that each data chunk is included in the corresponding blocks.
4. Validate the blocks containing the data chunks are part of the history of the ledger (i.e., previous blocks of LAST_FINALIZED_BLOCK).

Note that obtaining cryptographic proofs of steps 1-3 only traverses $O(\log(|tree|))$ elements in the Merkle tree, while for step 4, a standard hashed linked list of blocks may take linear time to cryptographically verify if a block is ahead of LAST_FINALIZED_BLOCK in the ledger. An improvement can be done by using a Merkle tree accumulator to store all the blocks as also adopted by Facebook Libra, and as a result, verifying the block relationship can be done in $O(\log(|blocks|))$ time.

- Read the metadata of the file from the state_trie_hash of LAST_FINALIZED_BLOCK, and validate all cryptographic proofs that the metadata is indeed in the state_trie.
- Read the chunk_info's from the chunk_info_trie_hash in the metadata, and validate all cryptographic proofs that the chunk_info's are included in the chunk_info_trie.
- Read the data chunk for each chunk_info, and verify that each data chunk is included in the corresponding blocks.
- Validate the blocks containing the data chunks are part of the history of the ledger (i.e., previous blocks of LAST_FINALIZED_BLOCK).

Note that obtaining cryptographic proofs of steps 1-3 only traverses $O(\log(|tree|))$ elements in the Merkle tree, while for step 4, a standard hashed linked list of blocks may take linear time to cryptographically verify if a block is ahead of LAST_FINALIZED_BLOCK in the ledger. An improvement can be done by using a Merkle tree accumulator to store all the blocks as also adopted by Facebook Libra, and as a result, verifying the block relationship can be done in $O(\log(|blocks|))$ time.

- Highly trustworthy storage for clients

: A writer can check the integrity of the written data according the cryptographic proofs of the blockchain without trusting the node that writes the data. For readers, assuming no writer challenges the proofs, any reader could use the proofs and verify such integrity and immunity of data from any node. Again, a reader only needs to trust the consensus and the ledger instead of a specific node. As a comparison, for traditional systems such as HDFS/GFS, we have to trust the systems are properly implemented and operated (e.g., a chunk of a file in a datanode in HDFS is corrupted (with checksum disabled) or the datanode is hacked, and the DFSClient has no way to verify the corrupted data after reading the chunk from the datanode).

Scalability

Scalability on Storage

To scale storage, a full node can be implemented as a cluster (server farms) and the block can be distributedly stored on the servers in the cluster. The cluster may or may not implement HA. If an HA feature is implemented, the full node itself may replicate the blocks to multiple servers in the cluster.

Scalability on Read

Similar to HDFS/GFS, scalability on read can be achieved in the way that a full node will respond to a data chunk read request with an IP address of the server that stores the actual data chunk in the cluster, and the following data read operation will be performed on that server instead of the full node itself.

Scalability on Write

All full nodes must reach the same view on all the blocks, and if the amount of requests of write operation is high and the blocks are large, synchronizing the blocks among the full nodes may be costly or even prohibited. To optimize write performance, we could have the following optimizations:

- (Transaction Submit Optimization)

. Instead of submitting the write transactions to the full node, a full node can instruct a DFSClient with a server in the cluster that receives the write transactions.

- (Transaction Broadcast Optimization)

. Conventional public blockchain network using Gossip protocol to broadcast transactions, which create extra traffic and is inefficiency. For a private/consortium blockchain, we could implement a pipeline protocol between the full nodes, where each full node assigns a server in its cluster to receive a write transaction and forwards the write transaction to another server in the next cluster in the pipeline similar to HDFS/GFS does.

- (Block Broadcast Optimization)

. When broadcasting a valid block, a block producer may assume most of the write transactions are already synchronized among full nodes, and thus it will only broadcast a compact version of the block where all transactions are replaced by their hashes in the compacted block. Upon receiving a compacted block, a node will check the existence of the transactions in the cluster, and if a transaction is missing, the node will instruct one of its servers to download the transaction from the peer. As a result, if multiple transactions are missing, downloading the missing transactions can be done in a parallel fashion.

- (Parallel Blockchains)

. Multi-chain/Sharding technology can be employed to increase the throughput and storage capacity (such as Boson Consensus). One extreme case is that each file is represented by a chain and a root chain only collects the hashes of the updated tips of the sub-chains for newly changed files.

Further Enhancements

- Mutual exclusion between writers upon open

: When opening a file for write or create operation, the operation will be prohibited if the file has already been open for write by another writer.

- Deletion

: Support delete(file_object). This operation prevents the file from opening, but the content of the file may still be accessible in the blockchain ledger.

- Access control

: The network can define the access (read/write) rights of each file.

- Directory

: Directory objects can be implemented in addition to file objects.

- Multiple writers

: We may allow multiple writers that append to a file concurrently. If multiple appends are called by multiple writers, we only ensure that the append is atomic (i.e., the appended data will not be interleaved by other data) if the data size is smaller than a threshold (e.g., chunk size).

- Write-any semantics

: Support write(file_object, buf, offset, len) operation. Note that any write operation with len greater than a threshold may not be atomic if multiple writers write the same part. Similarly, truncate(file_object) operation may also supported.

- Quota management

: The system can limit the space used by a user and prevent spamming.

- Garbage collect (GC)

: Garbage data on blocks may be produced if: * A file is deleted; or

- A write transaction overwrites the last chunk of previous write transaction, and thus the overwritten last chunk can be discarded.

To discard the garbage and reclaim the storage, a GC can be implemented by creating a new block that reclaims the space of the first unGCed block. This will replay the transactions of the first unGCed block (starting from genesis block) by updating the chunk_info trie accordingly without performing actual write. This will also increase the index of the first unGCed, which can be written in a field in the block header.

- A file is deleted; or
- A write transaction overwrites the last chunk of previous write transaction, and thus the overwritten last chunk can be discarded.

To discard the garbage and reclaim the storage, a GC can be implemented by creating a new block that reclaims the space of the first unGCed block. This will replay the transactions of the first unGCed block (starting from genesis block) by updating the chunk_info trie accordingly without performing actual write. This will also increase the index of the first unGCed, which can be written in a field in the block header.

Further Extensions

We may extend the idea to blockchain as a distributed key-value store (BaaDKYS).