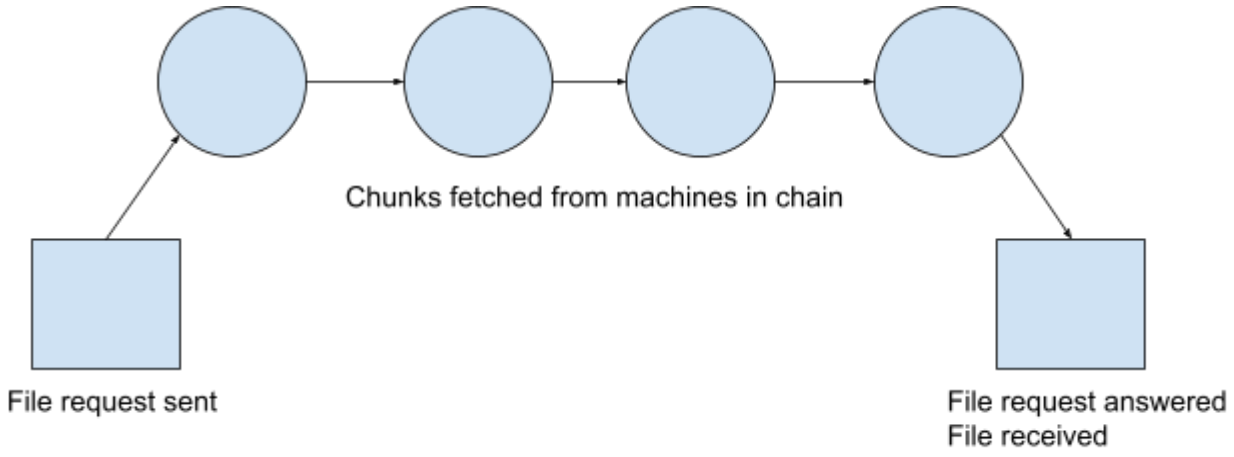
LACPP

Distributed Database

Linked Structure

Using linked refs to connect diff machines containing different files
All machines contain "Start Nodes" for all files
    Downside: Slow
    Upside: Redundant, Private, “Atomic” (as order exists in read/edit)
Has to impl TimeOut for slow/failed chunks



Direct Transactions (BROADCAST) (We should go with this)

Using a single key to call all machines, those who contain the key in their
FileStore return their respective chunk, other machines ignore/ignored
    Downside: No order, order data has to be implemented ***(KEY, (ORDER, CHUNK))***
        Not private (everyone listens for any files requested)
    Upside: Fast
Has to impl TimeOut for slow/failed chunks

Direct connections in erlang can be complicated, especially registering new nodes. Maybe it’s better to have a central registration server that all nodes communicate with?
One node says fetch(key), the server sends out a message to all other nodes with fetch(key) and returns the answers to the requesting node. Then the requesting node puts the chunks together to recreate the file.

BLOCKCHAIN DB ARCHITECTURE:

ABBREVIATIONS:

ONR - The order number for the chunk itself, where if we have n chunks to a file, the ONR's are unique integers in the set of {0, 1, 2, …, n-1}.

All nodes in the network should:

1. Contain a connection/ list of all other nodes in the network.
2. Contain a list containing one merkle hash for each of the files stored in the network.
3. Contain a local chunk DB file containing chunks stored in that machine including their respective merkle hash and *ONR's*, *(Merkle Key, (ONR, FileData)).*

Say that a file is stored/split in 4 chunks/ 4 machines.

ADD: When added we send out an add-request to all machines, the 4 fastest/ first to answer back to the add-request are chosen as candidates chunk holders.

1. The file is split in 4 (n) chunks, with a unique hash acting as a key for each of the chunks.
2. A merkle tree is then generated for all chunks, generating a master hash for all chunks, used for verification later on.
3. This merkle hash is then sent out to all nodes (machines) in the DB network
4. The file sending node must receive a "received" message from all the nodes in the network, one for receival of the merkle hash, containing the hash itself for cross checking with the newly generated merkle hash, and a file received from the file receiving machines.
5. The "received" messages are placed in a queue where the first 4 (n) to be placed in the *FIFO* queue are the nodes that are going to store that file. If one or more nodes do not answer, refer to the "**IN CASE OF FAILED CONNECTIONS**" section.
6. The file chunks are then sent to their respective 4 (n) locations in the network. expected to return a "chunk received" message. In case of timeout/ no message another node is popped from the queue in step 5 and chosen as the new file holder.
7. The user is then notified with a "file successfully added" message.

REMOVE: When we want to remove a file, we remove it based on the merkle hash.

1. A remove request is sent to all machines, where the merkle hash is removed from all machines, a "deleted" message needs to be received from all machines to confirm removal.
2. Machines that contain the file itself need to store a reference with the file *(Merkle Key, (ONR, FileData))*. These machines also in similar fashion send a "deleted" message to the deleter machine.
3. Now there shouldn't be any data of the file left in the network, either as binary chunk data or merkle hashes.

VIEW/GET: When we want to retrieve a file from the DB network, we use the merkle hash to do so.

1. A "get" message is sent to all the nodes in the network, where only the file holding nodes are required to answer back with a message containing the file and *ONR*, *(ONR, FileData)*. In case of failure/ all chunks not received, proceed with **REMOVE** and **DISCONNECTING A NODE**.

2. The user is then presented with a “failed to retrieve file/ file does not exist” message.

COUNT: When we want to count the files stored in the network, we just simply count the merkle hashes stored in the requesting node.

IN CASE OF FAILED CONNECTIONS: If no return message is received (after a given timeout) then that machine should be dropped from the network, as it is no longer functioning.

1. **DISCONNECTING A NODE:** File data/ merkle hashes remain still at all machines, data stored in the deleted node will not be considered at this point.
    a. A “disconnect” message is sent to all nodes in the network awaiting a “disconnected” message back from all machines. If a machine does not answer back from the “disconnect” message, repeat this step for that specific machine as well.
2. **RETRIEVING FILES FROM A DISCONNECTED NODE:** If we want to retrieve a file where a chunk existed in a node no longer in the network, we want to drop that file from the network.
    a. We begin to do a standard remove, using the merkle hash specific to that file, sent out to all nodes in the network (ref to **REMOVE**)
    b. The user is notified of this by a “file no longer exists” message.