THIS DOCUMENT IS A WORK IN PROGRESS DRAFT

OuiSync System Design

Design Goals

The following is a list of properties that may be desirable for a file synchronization tool. Not all of these properties must be achieved (some of them may even be mutually exclusive), but it is good to keep them in mind when considering different design options. More such interesting properties are listed in CryFS/4.2.

1. Encryption at rest & ability to sync without decrypting [decrypt-at-rest];
2. File names, file contents but also file sizes and structure of the directory tree should be hidden from replicas not possessing an encryption key [no-fingerprinting];
3. Losing some blocks shouldn't destroy the whole repository [robust]. This is mainly important if replicas download blocks in random order, where in such case it could be that the replica downloads a file, but not yet a directory where the file is located.;
4. History is not preserved by default [no-history];
5. History can be preserved explicitly to enable backups [backup];
6. A file stored in multiple directories should be represented only once [deduplication];
7. Option to disable encryption at rest per computer [fast-access];
8. A nice property to have here would be that if A, B and C are different versions of a particular object and B ≤ C, then A⊕B⊕C = A⊕C (where ⊕ represents a merge operation, and ≤ means "same or happened before"). [idempotent-merge];
9. Moving directories should be an O(1) operation [fast-move];
10. Receiving an update where a file is removed from one location but added to another location should not remove and then re-download that file [smart-move]
11. Replicas - whether they possess the decryption key or not - should download directories containing files before those files, also files should be downloaded from the beginning and not randomly [smart-download]. Note that this may help with [robust] in that with this we should avoid having "dangling" blocks which we don't know where they should be placed in the directory tree;
12. For most user-facing applications it is mostly desirable to switch from one branch to another continuously. That is, a user should see files being added and updated as soon as the data arrives. [partial-updates];
13. In contrast to [partial-updates], some applications may want to switch from one branch to another discretely. [discrete-updates].

Adversarial Model

General Idea

The purpose of OuiSync is to allow users to share and synchronize data across devices. At any point, the data is transferred and stored on the device is encrypted and no device needs to know the decryption key to perform the synchronization. In the following text, we'll call a replica any device that is set up to synchronize data with others. As such, OuiSync recognizes three types of replicas: (1) a blind replica is one which has access to the encrypted data, but does not necessarily possess a secret key to be able to read nor write the plaintext, (2) a reader replica is one that can decrypt the content and (3) a writer replica can decrypt and write.

The data in question is presented to the user as standard files and directories. Similarly as is done in CryFS, every file and every directory stored in OuiSync shall be sharded into relatively small (e.g. 16KB) blocks of a constant size. A linear set of blocks shall be called a blob. In CryFS, blobs are implemented using Left-Max-Data-Trees as described in CryFS/2. However, OuiSync shall differ in this regard, mainly to avoid issues related to concurrent write and deletion of incomplete blobs.

Another concept in OuiSync is that of a "snapshot". A snapshot represents a single user's view of the whole directory tree at a particular instance in time. Each modification of the directory tree (whether on a directory or a file) results in a new snapshot. We'll use the term "branch" to refer to the latest snapshot on a particular replica. Similarly as with blocks and blobs - but unlike e.g. git - the concept of a snapshot and a branch is internal to OuiSync these will not be exposed to the user through the UI. That is, whenever there is more than a single branch, the user shall see their union as explained in the Conflicts and Resolution section below.

It shall be an important invariant throughout this document that a single writer replica must not create divergent branches, though each replica will keep track of as many divergent branches as possible (implying: at most one per writer replica). The data structure used to keep track of all the branches inside one replica is called an "index".

Whenever two peers connect, they exchange their indices. These may be big in size, so the implementation should take care to only exchange the differences. Only once an index is downloaded and stored on a replica in its entirety, can each peer start downloading missing blocks. This is because both the reader and writer replicas need access to the tree information encoded in an index before they can perform certain operations.

Once an index is downloaded, it is then merged with the one the downloading replica already has. After doing so, a blind replica may delete all branches (and blocks in them that are not present in other branches) whose version vectors "happened before" version vectors from any other branch. We'll refer to this process as garbage collection on a branch. A blind replica may also decide to not delete such branches if it is set up to store backups.

Upon the index merge operation, a blind replica shall start downloading any missing blocks. If the replica is also a writer, then it has access to the tree structure of each branch and may also start performing per-file and per-directory garbage collection as described in the Garbage Collection section below.

Block Identification

To identify blocks, OuiSync uses block IDs. There are currently two different variants of these IDs under consideration. One where the block ID is calculated as a hash digest of the block, and the other where the ID is an array of random bytes. We'll refer to the former as HashBlockId and to the latter as RandomBlockId.

Both have their pros and cons. The main advantage to using HashBlockId is that it yields [deduplication] for free, i.e. blobs that occur multiple times in the repository shall be stored only once. However, calculating block hashes is a CPU intensive operation. Another disadvantage of using HashBlockIds is that care needs to be taken not to leak information about what is stored inside the block. That is, if an adversary is able to guess what the block stores - without additional precaution - he/she would be able to confirm it using the block's ID.

Finally, even if the above precautions are addressed, we'll learn in the next section that block IDs are stored on replicas together with a "locator" that holds information about where such a block is located with respect to other blocks. This information is normally hidden, but if hash object IDs are used, then due to [deduplication] the index structure would need to list each block ID N-times if N is the number of occurrences of that particular block.

RandomBlockIds don't have the above mentioned disadvantages. Generating a random ID is faster (assuming a good random number generator, but not the hardware based one). The one disadvantage is that [deduplication] would either need to be sacrificed or explicit calculation would need to be performed to support it.

Given the above pros and cons, OuiSync's design has settled on using the block IDs based on random numbers.

Index

As mentioned earlier, and similarly as in CryFS, OuiSync has a concept of Blobs which represent files and directories. In CryFS, blobs are implemented as Left-Max-Data trees where each inner node as well as each leaf node is a block. The inner nodes of such Left-Max-Data Tree contain block IDs that serve as pointers to the children nodes.

Additionally, when a blob represents a directory, such a directory contains a list of file names together with block IDs "pointing" to blocks that are roots of other blobs.

In Ouisync, due to its P2P nature where writer replicas may create concurrent modifications, this approach is not sufficient (TODO: elaborate?). Instead, Ouisync introduces a separate data structure called “Index”.

Index keeps track of divergent branches, version vectors associated with each such branch, all the block IDs per branch as well as the tree (or DAG, in case BlockID == HashBlockID) structure of each branch. In addition, the structure must be encoded in the index in such a way that only reader replicas can access this information.

Blobs are encoded as a linear sequence of blocks. Blocks contain only the actual content of the blob and not its structure. Thus unlike in CryFS, there are no inner nodes. The structure is instead encoded in the index.

Additionally, the index also encodes the directory structure of the repository. That is, the first block of a blob is associated with the ID of its parent directory blob.

Note: the directory structure is also encoded in the directory blob itself (as explained in more detail in the Directory Blobs section) so it might seem redundant to encode it in the index too. The reason for this is to identify orphaned blocks. Imagine we have some files with missing blocks (maybe the download was interrupted due to the sending replica disconnecting). If we wanted to delete a directory which contains those files then without the additional information in the index, we might have no way to know that the blocks we have belong to those files. Then deleting the directory would leave those blocks orphaned/dangling. But by having the information in the index, we can always trace the ancestors of those blocks and thus know when to delete them.

In the following, we will use the SQL language to describe the index, but we note that SQL is not necessary and this document uses it mostly for its expressiveness. In particular, the index structure can be expressed as the following two tables:

```
CREATE TABLE Branches (
       versions       VersionVector,    // Uniquely identifies a branch
       writer         WriterReplicaId,  // Identifies the branch
       root_block_id  BlockId,
       merkle_root    HashDigest        // To be explained in later sections
)

CREATE TABLE Index (
       block_id    BlockId,
       locator     HashDigest
)
```

TODO: An index should also keep track of missing blocks.

Most of the table columns are self explanatory (TODO: perhaps they are not and should be elaborated?). The remaining column that needs explanation is the `locator`. This field exists in the index structure to encode the tree information of a branch. Its semantics is akin to using block parent pointers, but without leaking the information to non-reader replicas.

Locator

Block IDs identify blocks but they contain no information about where in the filesystem the block is located (that is, which blob the block belongs to or which directory a blob belongs to). Thus an additional identification scheme needs to exist to answer such questions. This is what “Locators” are for.

While block ID changes when the block it identifies changes, locator never changes. On the other hand, block id is globally unique (across all replicas and snapshots) where the locator is unique only within a single snapshot.

Locator can be described as the following “sum type”:

```
Locator = Root
        | Head(HashDigest, UInt32)
        | Trunk(HashDigest, UInt32)
```

That is, a locator can be in of the following three variants:

- **Root**: locator pointing to the root block, that is, the “head” block of the root directory. The root locator can be thought of as a special case of “Head” locator which has no parent and the sequence number is implicitly 0.
- **Head**: locator pointing to the head (that is, first) block of a blob. It contains the hash of its parent locator and the sequence number. The parent locator is the locator of the parent directory of the blob (that is, the locator of the head block of the parent directory) and the sequence number identifies the blob among other blobs belonging to the same directory.
- **Trunk**: locator pointing to a block that is not a head block. The parent hash is of the corresponding head locator and the sequence number identifies the block among other blocks of the same blob and also defines their order.

Given a locator *H* representing a directory (that is, locator of the head block of the directory) and a sequence number *i* of an entry in that directory, the locator of that entry (that is, locator of the head block of that entry) can be obtained as:

```
Locator::Head(HASH(H), i)
```

Similarly, given a locator *B* of a blob (head block of the blob) and the sequence number *i* (where *i > 0*) of a block of that blob, the locator of that block can be obtained as:

```
Locator::Trunk(HASH(H), i)
```

This way, the structure of the whole repository can be retrieved. However, to not reveal this structure to agents that don’t possess the secret key, locators are not stored in the index directly. Instead they are encoded in the following way:

```
HashDigest encode_locator(SecretKey key, Locator locator)
```

```
{
        // Using a hash of key to prevent extension attacks.
        // TODO: Use HMAC instead?
        return HASH(HASH(key) ++ HASH(locator));
}
```

Where the hash of locator is calculated as follows:

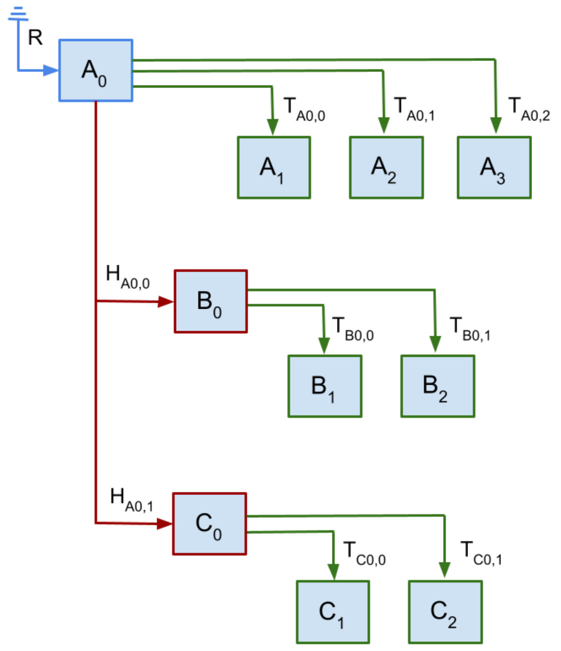
```
hash(Locator::Root)                = 0
hash(Locator::Head(parent, seq))  = HASH(parent ++ seq ++ 0)
hash(Locator::Trunk(parent, seq)) = HASH(parent ++ seq ++ 1)
```

Block pointer retrieval

The index needs to provide a way to retrieve a block id for a given locator

```
BlockId Index::get_block_id(SecretKey key, Locator locator) {
        HashDigest encoded_locator = encode_locator(key, locator);
        return query("SELECT block_id FROM Index WHERE locator = "
                ++ encoded_locator).at(0);
}
```

Doing so lets us represent the directory tree structure as well as the linear order of blocks inside each blob. This is depicted in the following example diagram.



Index:

Block	Locator	Type
A_0	R	Root
A_1	$T_{A0,0}$	Trunk
A_2	$T_{A0,1}$	Trunk
A_3	$T_{A0,2}$	Trunk
B_0	$H_{A0,0}$	Head
B_1	$T_{B0,0}$	Trunk
B_2	$T_{B0,1}$	Trunk
C_0	$H_{A0,1}$	Head
C_1	$T_{C0,0}$	Trunk
C_2	$T_{C0,1}$	Trunk

Example with three blobs A, B, C. The blob A consists of blocks [A_0,A_1,A_2,A_3], blob B of blocks [B_0,B_1,B_2] and block C of blocks [C_0,C_1,C_2]. Blob A represents a directory which itself contains the blobs B and C.

Directory Blobs

Reader and writer replicas can see individual blobs in plain text. There are two types: a file blob and a directory blob (possibly more in the future). Of these two, the file blob is the simpler one, it consists of a header containing the file size, permissions and a time stamp. The directory blob must represent a list of file names present in a directory represented by this blob, as well as locators pointing to the individual blobs (in practice only the sequence numbers need to be stored because the locators can be then computed from the directory locator and the seq. number).

Due to the distributed nature of OuiSync, where concurrent changes to files and directories are possible, the directory structure needs to be more elaborate. In particular, whenever two or more users modify a file in a directory concurrently no information must be lost, and upon reconciliation both versions need to be presented to a user for manual conflict resolution.

With this in mind, the directory structure is essentially a map defined as follows:

```
Directory     = Map<EntryName, EntryVersions>
EntryVersions = Map<WriterReplicaID, {VersionVector, UInt32 /* blob index */}>
```

Here, the EntryName is simply a string representing the name of the file or directory entry in this directory. As described above, each such entry may have multiple versions. Note however, that there may be at most one per writer replica. This enforces the invariant that no replica can create a version of a branch that is divergent to any other version from the same writer.

The type itself in this case does not - however - prevent from having an entry in the directory written by two different writer replicas, where the version vector of one "happened after" the other. Such invariant needs to be preserved by the writer replica whenever it writes the directory. Which may be either when the directory is being modified, or when the garbage collection runs on it.

Conflicts and Resolutions

Notice that due to the possibility of having multiple branches, as well as the possibility of having multiple versions of the same directory entry, the relationship between a file system's path and a blob is not one-to-one. Therefore, the triple (*<Branch>,<Path>,<Version>*) does identify a blob uniquely.

In the following we'll use the notation *<Branch>:<Path>:<Version>* to represent the above triple. We'll further use _*:<Path>:*_ to represent the set of all blobs corresponding to the same path across all branches and all versions present in any of those branches.

Due to this non one-to-one relationship between a file system path and a blob, it will sometimes be necessary to present the user with modified directory entry names to let them unambiguously select which blob they want to read or modify. We'll call the *internal path* the path by which blobs are referred to internally within OuiSync's data structures. And we'll call the *external path* the path that is presented to the user.

Example: Say there is a branch B and a file with internal path /foo/bar.txt which has two divergent versions (1,0) and (0,1). This means that there are two blobs in the set B:/foo/bar.txt:* and their external path shall be /foo/bar.txt-conflict-1 and /foo/bar.txt-conflict-2 (the actual extension may differ in the final implementation).

Whenever the set *:<Path>:* contains only a single blob, the external path equals to the internal one (i.e. <Path>). However when the set *:<Path>:* contains two or more blobs, we need to consider three cases:

1. the set contains only directory blobs
2. the set contains only file blobs
3. the set contains a mix of file and directory blobs

In the case #1 the external path will also equal to the internal path. And whenever the user requests the listing of such a set of directory blobs, a listing of each such blob shall be taken and a union of all the results shall be presented to the user.

The case #2 where the set *:<Path>:* contains two or more file blobs shall be presented to the user as in the above example. That is, a unique extension shall be appended to each blob's internal path.

Finally, when the set *:<Path>:* contains a mix of file and directory blos (the case #3), we'll split the set to two subsets directories(*:<Path>:*) and files(*:<Path>:*). Then one unique extension shall be appended to <Path> to represent all the directory blobs in directories(*:<Path>:*) as in case #1, and one unique extension shall be appended to each file in the in the files(*:<Path>:*) set as in case #2.

TODO: Tombstones & conflicts with them.
TODO: A conflict is resolved by deleting divergent versions.
TODO: When there are multiple versions of an entry and one of those versions was created by a writer replica, that replica can only modify that particular version. This is to avoid having multiple divergent versions created by the same replica. This is also enforced by the type of the Directory blob as described in the Directory Blobs section.

Garbage Collection

Garbage collection is the act of removing the directory entry B_i:<Path>:V_k whenever there exists an entry B_j:<Path>:V_l and $V_k < V_l$. Or in other words: removing an entry E whenever there exists another entry E' with the same internal path, such that the version vector of E' "happened after" the version vector of E.

An invariant that we want to preserve is such that whenever we have two or more versions of the same entry in a single branch: B:<Path>:V_i and B:<Path>:V_j it must be that V_i and V_j are divergent (i.e. $V_i \nless V_j$ and $V_j \nless V_i$ when $i \neq j$).

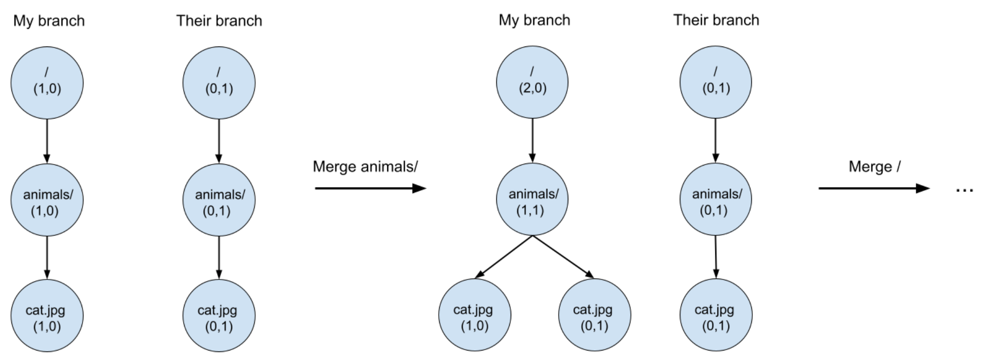
Whenever a writer replica modifies a directory entry B:<Path>:V it will always modify the version vector V by incrementing the counter corresponding to this replica's ID. Therefore resulting in the new entry B:<Path>:V' where V < V' and thus the old entry can and must be removed right away.

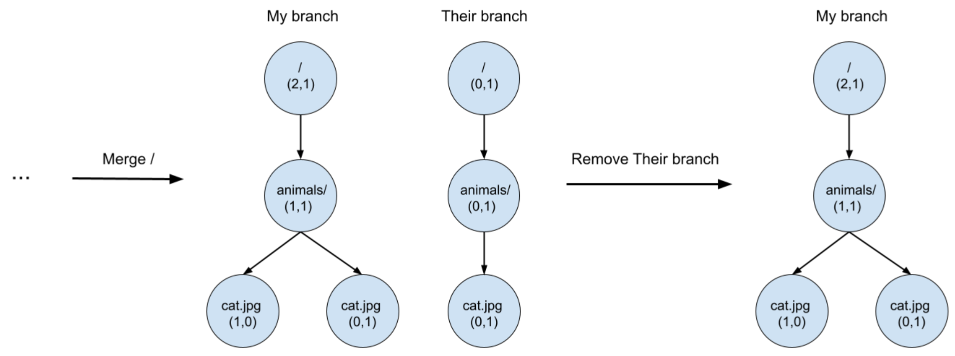
To garbage collect an entry $\{B_i,B_j\}$:<Path>:$\{V_i,V_j\}$ where $i \neq j$ and $V_i < V_j$ we'll need to be more elaborate. If there indeed exist two branches B_i and B_j on a single replica, it must be the case that their roots must have divergent commits, otherwise one of the branches would have been deleted by even a blind replica. When doing garbage collection, replicas need to find all such elements across all divergent branches, update them to their latest version on the branch where that replica has write access and finally update the version vector of the root of the branch to indicate that it "happened after" roots of the other branches. The last step lets blind replicas delete all the other branches which are no longer divergent. We will refer to this process as *branch merging*.

If done as described above, however, branch merging may take too long on large repositories. To make the matter worse, while branch merging is taking place, another replica may send this replica an update on its branch which would trigger another branch merging round.

To alleviate this problem, we'll do *directory merging* as well. The idea is to be able to make merging incremental as well as be able to share these intermediate results with other replicas.

We say that a directory needs to be merged, whenever there are two or more distinct branches that contain the directory in question but the version vectors of this directory across those branches don't match. Furthermore, we establish that a directory can be merged only when all its children directories have been merged as well. Once two directories D_{my} and D_{their} are merged, the resulting directory's R version vector is set to $\forall x \in$ WriterReplicaID: $v_{new}[x] = \max(v_{my}[x], v_{their}[x])$. Finally, the content of the directory R is taken to be the union of the contents of directories D_{my} and D_{thier}.





Things to note:

- Directory merging is a commutative operation. This is important because irrespective of where the merge happens, we want to end up with the same result.
- Given that child directories must have already been merged, it is enough to take the local version of a child directory into R.
- Files are never automatically merged when having divergent version vectors.

Upon each inner (non root) directory merge, the version vector of the root directory is updated by increasing the counter corresponding to "this" replica by one. This will let other replicas know that a change has happened and they can download the new partially merged index and possibly remove some garbage collected blocks.

The above does not happen for root directories. Instead, their version vectors are simply updated as per normal directory merge also described above. Once this is done, all other branches that haven't been updated during the root directory merge operation shall have version vectors that "happened before" the version vector of "this" branch and thus may be removed by all types of replicas.

TODO: Tombstones

Index Synchronization

One problem this document hasn't yet addressed is about how block membership to a branch shall be represented in OuiSync's internal structures. That is, we need to design a data structure that will give us answers to the following questions:

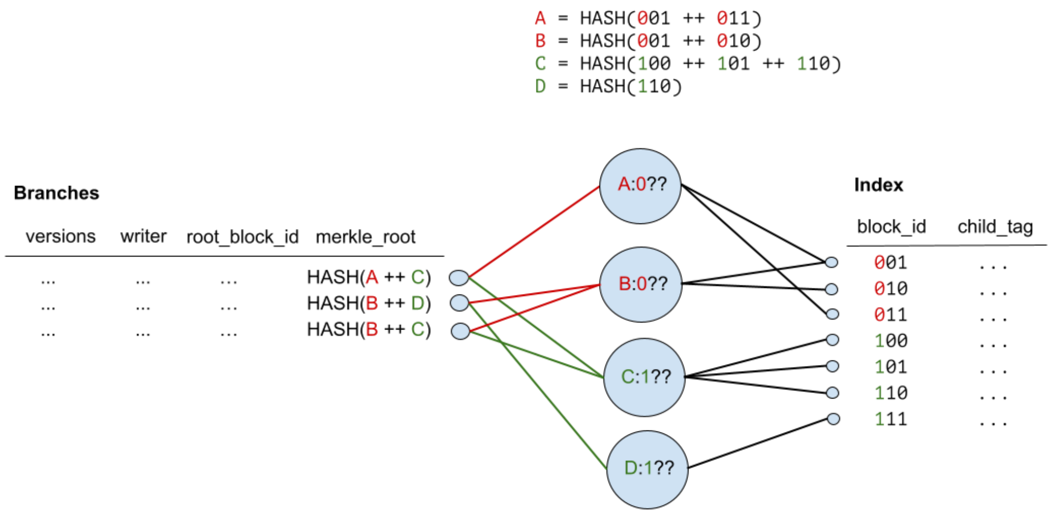
1. What branch does a particular block belong to?
2. What blocks belong to a particular branch?

The answer to the question #1 may be useful when trying to determine if a block can be removed completely from the index or if a replica wishes to restrict access to only certain branches (e.g. if it is a backup replica and only lets others access the most recent divergent branches). The answer to the question #2 is more evident and includes cases such as when a replica wishes to delete a branch or when one replica needs to communicate to another what blocks are in a particular branch.

In addition, this block ⇆ branch membership structure must be fast to create new branches out of old ones as well as allow fast communication of branch differences between peers. To achieve these goals, we'll use the Merkle-Tree data structure where the leaves shall point to the block IDs in the `Index` table, while the root hashes of the Merkle-Tree shall be stored in the `Branches` table as the `merkle_root` column.

A node in a Merkle-Tree must have a fixed size of children. This size should be equal to 2^K for some number K ≥ 1. When deciding which leaf of the Merkle-Tree a particular block ID should go to, we start at the root and pick the child corresponding to the first K bits of that ID. We continue this process on that child node with the next K bits of the ID. Repeating the process until we reach the leaf node.

A Merkle tree may have a fixed or a variable number of layers, it is to be decided which of the two approaches will be adopted by OuiSync, but taking K to be 8 and the number of layers to be 3 seems like a reasonable default. Assuming we want fewer than 256 of the expected number of blocks per child node and blocks of size 32KB, one repository should be able to handle up to ~137TB of data.



Example of the three Merkle-Tree data structures between the tables `Branches` and `Index`. These Merkle-Trees have only a single layer and each non leaf node has exactly two children.

Assuming OuiSync will use a SQL database to store the data, these sets of Merkle-Trees may be modelled using another table as follows:

```
CREATE TABLE MerkleForest (
        parent HashDigest,
        bucket Optional<UInt32>, // ∈ [0, 2^K) if the child is Merkle node, NIL
otherwise
        child  Either<HashDigest, BlockID>
)
```

Then a block inserting operation may look like so (note that the SQL language likely offers a richer and more optimized syntax):

```
HashDigest insert_block(HashDigest parent, BlockID block, UInt32 layer = 0)
{
        if (layer == MERKLE_TREE_DEPTH) {
                Set<BlockID> blocks =
                        query("SELECT child FROM MerkleForest WHERE parent = " + parent);

                blocks.insert(block);

                HashDigest new_parent = combined_hash(blocks);

                for (BlockID block_ : blocks) {
                        query("INSERT INTO MerkleForest VALUES "
                                "(" + new_parent + ", NIL," + block_ + ")");
                }

                return new_parent;
        }
        else {
                UInt32 bucket = nth_prefix(layer, block);

                Map<Bucket, HashDigest> buckets =
                        query("SELECT bucket, child FROM MerkleForest WHERE "
                                "parent = " + parent);

                buckets[bucket] = insert_block(buckets[bucket].front(), block, layer+1);
                HashDigest new_parent = combined_hash(buckets);

                for (auto [bucket_, child] : buckets) {
                        query("INSERT INTO MerkleForest VALUES "
                                "(" + new_parent + "," + bucket_ + "," + child + ")");
                }

                return new_parent;
        }
}
```

Note that after inserting a block as per the above function, the nodes of the old Merkle-Tree still exist in the `MerkleForest` table and must be removed explicitly if the root of the tree is not present in any other branch.

Using Merkle-Trees as just described gives us an efficient way for peers to communicate information about which blocks belong to any particular branch. That is, instead of one peer sending to the other all the `(block_id, child_tag)` pairs corresponding to a certain branch, the receiving peer may instead ask questions such as "send me all children of merkle-tree node X", to which the sending peer will reply with either a list of merkle tree hashes of the child nodes, or some subset of `(block_id, child_tag)` pairs to which the node X points.

Tracking Missing Blocks

Once a snapshot index is downloaded, the replica should start downloading blocks (data associated with BlockID) for which the BlockIds are listed in the snapshot index but which are not present in the block store. The particular snapshot may have a lot of blocks missing and so this "download all missing blocks" operation may take a long time. Therefore we need a way to quickly find out whether a snapshot still has some blocks missing, and if so, find out which ones.

One way to do this would be to annotate each node in the branch forest with a number indicating how many leaf nodes are missing in the tree generated by that node. Let's call this field *missing_block_count*. To find a missing BlockId in a snapshot, the replica would look at the snapshot root node and if it indicated that it has at least one block missing, it would recurse to its child nodes.

However, this is not enough. It is because one BlockId may be simultaneously missing from more than one snapshot and thus updating a single snapshot each time a new block arrives would leave the other snapshots in an inconsistent state. A way around this would be to start at the leaf corresponding to the BlockId of the newly acquired block and then recursively update its parents with the new number of missing blocks.

Unfortunately, because the number of snapshots isn't limited, updating all of them in one go could be a prohibitively long task. On top of that, if the replica crashed during this process, it would likely - again - leave the index in an inconsistent state.

To address this problem, the replica shall store the block ID of the newly downloaded block in persistent storage in a table named *dirty_leaves* to indicate that ancestors (the reverse tree generated by the leaf corresponding to the block ID) need to have their *missing_block_count* field updated. In the following text, we'll call any node *dirty* if it is either a leaf node that is contained in *dirty_leaves* or has a *dirty* child.

Once a block ID has been successfully stored in the *dirty_leaves* table, the replica may start updating its parent nodes recursively one snapshot at a time (i.e. in a reverse depth-first-order). While doing so, at each node the replica shall look into the *missing_block_count* field of its children and set its own *missing_block_count* field to the sum of the results. Note that while doing so, some of the children nodes may themselves be dirty, this - however - is not a problem because eventually this node shall be recalculated again through those dirty children.

The above mechanism may further be improved by renaming the *dirty_leaves* table to *dirty_nodes* as well as modifying its semantics such that it would include all nodes (i.e. not only leaves) which themselves are dirty, but whose all children are not dirty. This modification would allow for a finer recovery from a crash as replicas would not need to recalculate all snapshots containing a particular dirty leaf if those snapshots have already been recalculated prior to the crash.

Networking

Choosing connections

TODO:

- Group by type (IP, Bluetooth,...), select one from each group
- Rabin's choice coordination algorithm

Solved Problems

Here is a list of problems encountered while the design was being created, the final design is then a one where these problems have been eliminated.

Incomplete erase {incomplete-erase}

In a design such as the one in CryFS where block's membership in a blob is decided by other such parent blocks that belong to that particular blob, it is possible to end up with "dangling" blocks: ones that are neither referred to by other blocks, nor listed in the set of deleted blocks as described in CryFS/4.4.1.

Such dangling blocks can be created when the application tries to delete a blob which hasn't been fully downloaded. In particular, when some of the non-leaf nodes have been downloaded. This would imply that there may exist blocks which should be deleted but aren't as well as not put into the remove set.

An apparent - though invalid - solution would seem to be a one where the application is forced to wait until the whole blob has been downloaded. But this has two issues, first being that the blob in question may be prohibitively big, and the other that there may actually no longer be a peer that can provide us with the remaining blocks.

The current proposal for the solution encodes the whole repository structure in the index using the *child_tag* as described in the Index section of this document.

Remove set grows monotonically {remove-list}

CryFS uses a "remove set" described in CryFS/4.4.1 to mark which blocks have been removed from the repository. One disadvantage of this approach is that such a set can only grow. Therefore frequently adding and removing large files to and from the repository may result in a prohibitively large set.

OuiSync's approach is to instead use an Index structure that keeps track of all the blocks present in the repository.